JBoss Enterprise Application Platform 5

HornetQ User Guide

for use with JBoss Enterprise Application Platform 5

Andy Taylor

JBoss Enterprise Application Platform 5 HornetQ User Guide for use with JBoss Enterprise Application Platform 5 Edition 5.1.1

Author	Andy Taylor	ataylor@redhat.com
Editor	Laura Bailey	lbailey@redhat.com
Editor	Jared Morgan	jmorgan@redhat.com
Editor	Rebecca Newton	rnewton@redhat.com

Copyright © 2010 Red Hat, Inc.

The text of and illustrations in this document are licensed by Red Hat under a Creative Commons Attribution–Share Alike 3.0 Unported license ("CC-BY-SA"). An explanation of CC-BY-SA is available at *http://creativecommons.org/licenses/by-sa/3.0/*. In accordance with CC-BY-SA, if you distribute this document or an adaptation of it, you must provide the URL for the original version.

Red Hat, as the licensor of this document, waives the right to enforce, and agrees not to assert, Section 4d of CC-BY-SA to the fullest extent permitted by applicable law.

Red Hat, Red Hat Enterprise Linux, the Shadowman logo, JBoss, MetaMatrix, Fedora, the Infinity Logo, and RHCE are trademarks of Red Hat, Inc., registered in the United States and other countries.

Linux® is the registered trademark of Linus Torvalds in the United States and other countries.

Java® is a registered trademark of Oracle and/or its affiliates.

XFS® is a trademark of Silicon Graphics International Corp. or its subsidiaries in the United States and/or other countries.

MySQL® is a registered trademark of MySQL AB in the United States, the European Union and other countries.

All other trademarks are the property of their respective owners.

A guide to HornetQ for JBoss Enterprise Application Platform 5 and its patch releases.

Preface **xi**
1. Document Conventions xi
1.1. Typographic Conventions xi
1.2. Pull-quote Conventions xii
1.3. Notes and Warnings xiii
2. Getting Help and Giving Feedback xiii
2.1. Do You Need Help? xiii
2.2. Give us Feedback xiv

1. Introduction **1**

2. Migrating to HornetQ **3**
2.1. Before You Migrate 3
2.1.1. Back up relevant data 3
2.2. Application Code 4
2.3. Installing HornetQ 4
2.4. Server Configuration Migration 4
2.5. Migrating JMS-administered Objects and Bridges 6
2.6. Other Configuration in JBoss Messaging 9
2.7. Migrating Existing Messages 9
2.8. Applications that use management APIs 9

3. Getting Started with HornetQ **11**
3.1. Starting The Server 11
3.1.1. HornetQ and JBoss Enterprise Application Platform 11
3.2. Running the Examples 11
3.2.1. The JMS examples 11
3.2.2. The Java EE Examples 13

4. Messaging Concepts **17**
4.1. Messaging Concepts 17
4.2. Messaging styles 17
4.2.1. The Message Queue Pattern 17
4.2.2. The Publish-Subscribe Pattern 18
4.3. Delivery guarantees 18
4.4. Transactions 18
4.5. Durability 18
4.6. Messaging APIs and protocols 19
4.6.1. Java Message Service (JMS) 19
4.6.2. System specific APIs 19
4.6.3. STOMP 19
4.7. High Availability 19
4.8. Clusters 20
4.9. Bridges and routing 20

5. Core Architecture **21**

6. Using the Server **23**
6.1. Library Path 23
6.2. System properties 23
6.3. Configuration files 23
6.4. The main configuration file. 24

7. Using JMS **25**
7.1. A Simple Ordering System 25
7.2. JMS Server Configuration 25
7.3. Connection Factory Types 26

7.4. JNDI configuration 26
7.5. The code 27
7.6. Directly instantiating JMS Resources without using JNDI 28
7.7. Setting The Client ID 29
7.8. Setting The Batch Size for DUPS_OK 29
7.9. Setting The Transaction Batch Size 30

8. Using Core **31**
8.1. Core Messaging Concepts 31
8.1.1. Message 31
8.1.2. Address 31
8.1.3. Queue 32
8.1.4. ClientSessionFactory 32
8.1.5. ClientSession 32
8.1.6. ClientConsumer 32
8.1.7. ClientProducer 33
8.2. A simple example of using Core 33

9. Mapping JMS Concepts to the Core API **35**

10. The Client Classpath **37**
10.1. HornetQ Core Client 37
10.2. JMS Client 37
10.3. JMS Client with JNDI 37

11. Examples **39**
11.1. JMS Examples 39
11.1.1. Application-Layer Failover 39
11.1.2. Core Bridge Example 39
11.1.3. Browser 39
11.1.4. Client Kickoff 40
11.1.5. Client-Side Load-Balancing 40
11.1.6. Clustered Grouping 40
11.1.7. Clustered Queue 40
11.1.8. Clustered Standalone 40
11.1.9. Clustered Topic 40
11.1.10. Message Consumer Rate Limiting 40
11.1.11. Dead Letter 41
11.1.12. Delayed Redelivery 41
11.1.13. Divert 41
11.1.14. Durable Subscription 41
11.1.15. Embedded 41
11.1.16. HTTP Transport 41
11.1.17. Instantiate JMS Objects Directly 41
11.1.18. Interceptor 42
11.1.19. JAAS 42
11.1.20. JMX Management 42
11.1.21. Large Message 42
11.1.22. Last-Value Queue 42
11.1.23. Load Balanced Clustered Queue 42
11.1.24. Management 42
11.1.25. Management Notification 43
11.1.26. Message Counter 43
11.1.27. Message Expiration 43
11.1.28. Message Group 43
11.1.29. Message Group 43

11.1.30. Message Priority 43
11.1.31. No Consumer Buffering 44
11.1.32. Non-Transaction Failover With Server Data Replication 44
11.1.33. Paging 44
11.1.34. Pre-Acknowledge 44
11.1.35. Message Producer Rate Limiting 44
11.1.36. Queue 44
11.1.37. Message Redistribution 44
11.1.38. Queue Requester 45
11.1.39. Queue with Message Selector 45
11.1.40. Reattach Node example 45
11.1.41. Request-Reply example 45
11.1.42. Scheduled Message 45
11.1.43. Security 45
11.1.44. Send Acknowledgments 45
11.1.45. SSL Transport 45
11.1.46. Static Message Selector 45
11.1.47. Static Message Selector Using JMS 45
11.1.48. Stomp 45
11.1.49. Stomp Over Web Sockets 46
11.1.50. Symmetric Cluster 46
11.1.51. Temporary Queue 46
11.1.52. Topic 46
11.1.53. Topic Hierarchy 46
11.1.54. Topic Selector 1 46
11.1.55. Topic Selector 2 46
11.1.56. Transaction Failover With Data Replication 46
11.1.57. Transactional Session 46
11.1.58. XA Heuristic 47
11.1.59. XA Receive 47
11.1.60. XA Send 47
11.1.61. XA with Transaction Manager 47
11.2. Core API Examples 47
11.3. Java EE Examples 47
11.3.1. EJB/JMS Transaction 47
11.3.2. HAJNDI (High Availability) 47
11.3.3. Resource Adapter Configuration 47
11.3.4. Resource Adapter Remote Server Configuration 47
11.3.5. JMS Bridge 47
11.3.6. MDB (Message Driven Bean) 48
11.3.7. Servlet Transport 48
11.3.8. Servlet SSL Transport 48
11.3.9. XA Recovery 48

12. Routing Messages With Wild Cards **49**

13. Understanding the HornetQ Wildcard Syntax **51**

14. Filter Expressions **53**

15. Persistence **55**
15.1. Configuring the bindings journal 56
15.2. Configuring the JMS journal 56
15.3. Configuring the message journal 56
15.4. Configuring HornetQ for Zero Persistence 58
15.5. Import/Export the Journal Data 58

16. Configuring the Transport **59**
16.1. Understanding Acceptors 59
16.2. Understanding Connectors 59
16.3. Configuring the transport directly from the client side 60
16.4. Configuring the Netty transport 61
16.4.1. Configuring Netty TCP 61
16.4.2. Configuring Netty SSL 62
16.4.3. Configuring Netty HTTP 63
16.4.4. Configuring Netty Servlet 63

17. Detecting Dead Connections **67**
17.1. Cleaning up Dead Connection Resources on the Server 67
17.1.1. Closing core sessions or JMS connections that have failed to close 68
17.2. Detecting failure from the client side. 68
17.3. Configuring Asynchronous Connection Execution 69

18. Resource Manager Configuration **71**

19. Flow Control **73**
19.1. Consumer Flow Control 73
19.1.1. Window-Based Flow Control 73
19.1.2. Rate limited flow control 74
19.2. Producer flow control 75
19.2.1. Window based flow control 75
19.2.2. Rate limited flow control 77

20. Guarantees of sends and commits **79**
20.1. Guarantees of Transaction Completion 79
20.2. Guarantees of Non Transactional Message Sends 79
20.3. Guarantees of Non-Transactional Acknowledgments 80
20.4. Asynchronous Send Acknowledgments 80
20.4.1. Asynchronous Send Acknowledgments 80

21. Message Redelivery and Undelivered Messages **81**
21.1. Delayed Redelivery 81
21.1.1. Configuring Delayed Redelivery 81
21.2. Dead Letter Addresses 81
21.2.1. Configuring Dead Letter Addresses 82
21.2.2. Dead Letter Properties 82
21.3. Delivery Count Persistence 82

22. Message Expiry **85**
22.1. Message Expiry 85
22.2. Configuring Expiry Addresses 85
22.3. Configuring The Expiry Reaper Thread 85
22.4. Example 86

23. Large Messages **87**
23.1. Configuring the server 87
23.2. Setting the limits 87
23.2.1. Using Core API 87
23.2.2. Using JMS 88
23.3. Streaming large messages 88
23.3.1. Streaming over Core API 88
23.3.2. Streaming over JMS 89
23.4. Streaming Alternative 90
23.5. Cache Large Messages on client 90

23.6. Large message example 91

24. Paging **93**
24.1. Page Files 93
24.2. Configuration 93
24.3. Paging Mode 93
24.3.1. Configuration 94
24.4. Dropping messages 94
24.5. Blocking producers 94
24.6. Caution with Addresses with Multiple Queues 95

25. Queue Attributes **97**
25.1. Predefined Queues 97
25.2. Using the API 97
25.3. Configuring Queues Through Address Settings 98

26. Scheduled Messages **101**
26.1. Scheduled Delivery Property 101
26.2. Example 101

27. Last-Value Queues **103**
27.1. Configuring Last-Value Queues 103
27.2. Using Last-Value Property 103
27.3. Example 103

28. Message Grouping **105**
28.1. Using Core API 105
28.2. Using JMS 105
28.3. Example 106
28.4. Example 106
28.5. Clustered Grouping 106
28.5.1. Clustered Grouping Best Practices 107
28.5.2. Clustered Grouping Example 107

29. Pre-Acknowledge Mode **109**
29.1. Using PRE_ACKNOWLEDGE 109
29.2. Example 110

30. Management **111**
30.1. The Management API 111
30.1.1. Core Management API 111
30.1.2. JMS Management API 115
30.2. Using Management Via JMX 117
30.2.1. Configuring JMX 117
30.2.2. Example 118
30.3. Using Management Via Core API 118
30.3.1. Configuring Core Management 119
30.4. Using Management Via JMS 119
30.4.1. Configuring JMS Management 120
30.4.2. Example 120
30.5. Management Notifications 120
30.5.1. JMX Notifications 121
30.5.2. Core Messages Notifications 121
30.5.3. JMS Messages Notifications 121
30.5.4. Example 122
30.6. Message Counters 122
30.6.1. Configuring Message Counters 123

30.6.2. Example 123
30.7. Administering HornetQ Resources Using the Admin Console 123
30.7.1. JMS Queues 124
30.7.2. JMS Topics 125
30.7.3. JMS Connection Factories 125

31. Security **127**
31.1. Role based security for addresses 127
31.2. Secure Sockets Layer (SSL) Transport 128
31.3. Basic user credentials 129
31.4. Changing the security manager 129
31.5. JAAS Security Manager 130
31.5.1. Example 130
31.6. JBoss AS Security Manager 130
31.6.1. Configuring Client Login 130
31.7. Changing the username/password for clustering 131

32. Application Server Integration and Java EE **133**
32.1. Configuring Message-Driven Beans 133
32.1.1. Using Container-Managed Transactions 134
32.1.2. Using Bean-Managed Transactions 135
32.1.3. Using Message Selectors with Message-Driven Beans 136
32.2. Sending Messages from within Java EE components 136
32.3. MDB and Consumer pool size 137
32.4. Configuring the JCA Adaptor 138
32.4.1. Global Properties 139
32.4.2. Adapter Outbound Configuration 142
32.4.3. Adapter Inbound Configuration 143
32.5. Configuring the JBoss Application Server to connect to Remote HornetQ Server 144
32.5.1. Configuring Jboss 5 144
32.5.2. Configuring JBoss 5 MDB 147
32.6. High Availability JNDI (HA-JNDI) 147
32.7. XA Recovery 148
32.7.1. XA Recovery Configuration 148
32.7.2. Example 149

33. The JMS Bridge **151**
33.1. JMS Bridge Parameters 153
33.2. Source and Target Connection Factories 155
33.3. Source and Target Destination Factories 155
33.4. Quality Of Service Modes 155
33.4.1. AT_MOST_ONCE 156
33.4.2. DUPLICATES_OK 156
33.4.3. ONCE_AND_ONLY_ONCE 156
33.4.4. Time outs and the JMS bridge 156
33.4.5. Examples 157

34. Client Reconnection and Session Reattachment **159**
34.1. 100% Transparent session re-attachment 159
34.2. Session reconnection 159
34.3. Configuring reconnection/reattachment attributes 159

35. Diverting and Splitting Message Flows **161**
35.1. Exclusive Diverts 161
35.2. Non-exclusive Diverts 162

36. Core Bridges **163**

36.1. Configuring Bridges 163

37. Duplicate Message Detection **167**
37.1. Using Duplicate Detection for Message Sending 167
37.2. Configuring the Duplicate ID Cache 168
37.3. Duplicate Detection and Bridges 168
37.4. Duplicate Detection and Cluster Connections 169
37.5. Duplicate Detection and Paging 169

38. Clusters **171**
38.1. Clusters Overview 171
38.2. Server discovery 171
38.2.1. Broadcast Groups 172
38.2.2. Discovery Groups 173
38.2.3. Defining Discovery Groups on the Server 173
38.2.4. Discovery Groups on the Client Side 174
38.3. Server-Side Message Load Balancing 175
38.3.1. Configuring Cluster Connections 176
38.3.2. Cluster User Credentials 177
38.4. Client-Side Load balancing 178
38.5. Specifying Members of a Cluster Explicitly 179
38.5.1. Specify List of Servers on the Client Side 179
38.5.2. Specifying List of Servers to form a Cluster 180
38.6. Message Redistribution 181
38.7. Cluster topologies 182
38.7.1. Symmetric cluster 182
38.7.2. Chain cluster 182

39. High Availability and Failover **185**
39.1. Live - Backup Pairs 185
39.1.1. HA modes 185
39.1.2. Shared Store 185
39.2. Failover Modes 186
39.2.1. Automatic Client Failover 187
39.2.2. Getting Notified of Connection Failure 189
39.2.3. Application-Level Failover 190

40. Colocated and Dedicated Symmetrical Cluster Configuration **191**
40.1. Colocated Symmetrical Live and Backup Cluster 191
40.1.1. Colocated Live Server 194
40.1.2. Colocated Backup Server 198
40.2. Dedicated Symmetrical Live and Backup Clusters 203
40.2.1. Dedicated JCA Live Server 206
40.2.2. Dedicated JCA Backup Server 210
40.2.3. Dedicated Remote Server 211

41. Libaio Native Libraries **215**
41.1. Compiling the native libraries 215
41.1.1. Install requirements 215
41.1.2. Invoking the compilation 216

42. Thread management **217**
42.1. Server-Side Thread Management 217
42.1.1. Server Scheduled Thread Pool 217
42.1.2. General Purpose Server Thread Pool 217
42.1.3. Expiry Reaper Thread 218
42.1.4. Asynchronous IO 218

42.2. Client-Side Thread Management 218

43. Logging **221**
43.1. Logging With The JBoss Application Server 221

44. Intercepting Operations **223**
44.1. Implementing The Interceptors 223
44.2. Configuring The Interceptors 223
44.3. Interceptors on the Client Side 223
44.4. Example 223

45. Interoperability **225**
45.1. Stomp 225
45.1.1. Native Stomp support 225
45.1.2. Mapping Stomp destinations to HornetQ addresses and queues 225
45.1.3. STOMP and connection-ttl 225
45.1.4. Stomp and JMS interoperabilty 226
45.1.5. Stomp Over Web Sockets 227
45.1.6. StompConnect 227

46. Performance Tuning **229**
46.1. Tuning persistence 229
46.2. Tuning JMS 229
46.3. Other Tunings 230
46.4. Tuning Transport Settings 231
46.5. Avoiding Anti-Patterns 231

47. Configuration Reference **233**
47.1. Server Configuration 233
47.1.1. hornetq-configuration.xml 233
47.1.2. hornetq-jms.xml 240

A. Revision History **245**

Preface

1. Document Conventions

This manual uses several conventions to highlight certain words and phrases and draw attention to specific pieces of information.

In PDF and paper editions, this manual uses typefaces drawn from the *Liberation Fonts*[1] set. The Liberation Fonts set is also used in HTML editions if the set is installed on your system. If not, alternative but equivalent typefaces are displayed. Note: Red Hat Enterprise Linux 5 and later includes the Liberation Fonts set by default.

1.1. Typographic Conventions

Four typographic conventions are used to call attention to specific words and phrases. These conventions, and the circumstances they apply to, are as follows.

`Mono-spaced Bold`

Used to highlight system input, including shell commands, file names and paths. Also used to highlight keycaps and key combinations. For example:

> To see the contents of the file **`my_next_bestselling_novel`** in your current working directory, enter the **`cat my_next_bestselling_novel`** command at the shell prompt and press **`Enter`** to execute the command.

The above includes a file name, a shell command and a keycap, all presented in mono-spaced bold and all distinguishable thanks to context.

Key combinations can be distinguished from keycaps by the hyphen connecting each part of a key combination. For example:

> Press **`Enter`** to execute the command.
>
> Press **`Ctrl`**+**`Alt`**+**`F2`** to switch to the first virtual terminal. Press **`Ctrl`**+**`Alt`**+**`F1`** to return to your X-Windows session.

The first paragraph highlights the particular keycap to press. The second highlights two key combinations (each a set of three keycaps with each set pressed simultaneously).

If source code is discussed, class names, methods, functions, variable names and returned values mentioned within a paragraph will be presented as above, in **`mono-spaced bold`**. For example:

> File-related classes include **`filesystem`** for file systems, **`file`** for files, and **`dir`** for directories. Each class has its own associated set of permissions.

Proportional Bold

This denotes words or phrases encountered on a system, including application names; dialog box text; labeled buttons; check-box and radio button labels; menu titles and sub-menu titles. For example:

> Choose **System** → **Preferences** → **Mouse** from the main menu bar to launch **Mouse Preferences**. In the **Buttons** tab, click the **Left-handed mouse** check box and click

[1] https://fedorahosted.org/liberation-fonts/

> **Close** to switch the primary mouse button from the left to the right (making the mouse suitable for use in the left hand).
>
> To insert a special character into a **gedit** file, choose **Applications** → **Accessories** → **Character Map** from the main menu bar. Next, choose **Search** → **Find…** from the **Character Map** menu bar, type the name of the character in the **Search** field and click **Next**. The character you sought will be highlighted in the **Character Table**. Double-click this highlighted character to place it in the **Text to copy** field and then click the **Copy** button. Now switch back to your document and choose **Edit** → **Paste** from the **gedit** menu bar.

The above text includes application names; system-wide menu names and items; application-specific menu names; and buttons and text found within a GUI interface, all presented in proportional bold and all distinguishable by context.

Mono-spaced Bold Italic or ***Proportional Bold Italic***

Whether mono-spaced bold or proportional bold, the addition of italics indicates replaceable or variable text. Italics denotes text you do not input literally or displayed text that changes depending on circumstance. For example:

> To connect to a remote machine using ssh, type **ssh *username*@*domain.name*** at a shell prompt. If the remote machine is **example.com** and your username on that machine is john, type **ssh john@example.com**.
>
> The **mount -o remount *file-system*** command remounts the named file system. For example, to remount the **/home** file system, the command is **mount -o remount /home**.
>
> To see the version of a currently installed package, use the **rpm -q *package*** command. It will return a result as follows: ***package-version-release***.

Note the words in bold italics above — username, domain.name, file-system, package, version and release. Each word is a placeholder, either for text you enter when issuing a command or for text displayed by the system.

Aside from standard usage for presenting the title of a work, italics denotes the first use of a new and important term. For example:

> Publican is a *DocBook* publishing system.

1.2. Pull-quote Conventions

Terminal output and source code listings are set off visually from the surrounding text.

Output sent to a terminal is set in **mono-spaced roman** and presented thus:

```
books        Desktop   documentation  drafts  mss    photos   stuff  svn
books_tests  Desktop1  downloads      images  notes  scripts  svgs
```

Source-code listings are also set in **mono-spaced roman** but add syntax highlighting as follows:

```
package org.jboss.book.jca.ex1;

import javax.naming.InitialContext;
```

```
public class ExClient
{
   public static void main(String args[])
       throws Exception
   {
      InitialContext iniCtx = new InitialContext();
      Object         ref    = iniCtx.lookup("EchoBean");
      EchoHome       home   = (EchoHome) ref;
      Echo           echo   = home.create();

      System.out.println("Created Echo");

      System.out.println("Echo.echo('Hello') = " + echo.echo("Hello"));
   }
}
```

1.3. Notes and Warnings

Finally, we use three visual styles to draw attention to information that might otherwise be overlooked.

Note

Notes are tips, shortcuts or alternative approaches to the task at hand. Ignoring a note should have no negative consequences, but you might miss out on a trick that makes your life easier.

Important

Important boxes detail things that are easily missed: configuration changes that only apply to the current session, or services that need restarting before an update will apply. Ignoring a box labeled 'Important' will not cause data loss but may cause irritation and frustration.

Warning

Warnings should not be ignored. Ignoring warnings will most likely cause data loss.

2. Getting Help and Giving Feedback

2.1. Do You Need Help?

If you experience difficulty with a procedure described in this documentation, visit the Red Hat Customer Portal at *http://access.redhat.com*. Through the customer portal, you can:

- search or browse through a knowledgebase of technical support articles about Red Hat products.
- submit a support case to Red Hat Global Support Services (GSS).
- access other product documentation.

Red Hat also hosts a large number of electronic mailing lists for discussion of Red Hat software and technology. You can find a list of publicly available mailing lists at *https://www.redhat.com/mailman/listinfo*. Click on the name of any mailing list to subscribe to that list or to access the list archives.

2.2. Give us Feedback

If you find a typographical error, or know how this guide can be improved, we would love to hear from you. Submit a report in Bugzilla against the product **JBoss Enterprise Application Platform 5** and the component **doc-HornetQ_User_Guide**. The following link will take you to a pre-filled bug report for this product: *http://bugzilla.redhat.com/*[2].

Fill out the following template in Bugzilla's **Description** field. Be as specific as possible when describing the issue; this will help ensure that we can fix it quickly.

```
Document URL:

Section Number and Name:

Describe the issue:

Suggestions for improvement:

Additional information:
```

Be sure to give us your name so that you can receive full credit for reporting the issue.

[2] https://bugzilla.redhat.com/enter_bug.cgi?product=JBoss%20Enterprise%20Application%20Platform%205&component=doc-HornetQ_User_Guide&version=12&short_desc=Bug%20in%20HornetQ%20User%20Guide

Introduction

JBoss Enterprise Application Platform messaging is based on HornetQ, which is an Open Source Message Oriented Middleware (MoM) project developed by the JBoss Community.

The Red Hat certified HornetQ product differs from the community offering because it is tested to ensure reliability on a number of hardware platforms, and is backed by a dedicated support network for product updates and technical support.

HornetQ is a Java-based multi-protocol, clustered, asynchronous messaging system. It features a high-performance journaling system that offers considerable performance for persistent messaging implementations.

The High Availability (HA) functionality offered by HornetQ features automatic client failover in the event of server node failure.

Migrating to HornetQ

This chapter is a guide to migrating existing JBoss Messaging applications to HornetQ. This chapter should be used in combination with the *Installation Guide* provided with this release of JBoss Enterprise Application Platform, which includes instructions on installing an instance of JBoss Enterprise Application Platform with HornetQ.

2.1. Before You Migrate

Shut down client and server

You must shut down your client and server before attempting to migrate from JBoss Messaging to HornetQ.

HornetQ does not use a database

JBoss Messaging uses a database to store persistent data unless null persistence is specified. HornetQ uses its own high-performance journal system instead of a database, so your database does not need to be shut down for migration purposes.

2.1.1. Back up relevant data

It is important to back up all data used in your application and your JBoss Messaging server before you migrate. This section outlines the data recommended for backup before migration.

2.1.1.1. JBoss Messaging database tables

JBoss Messaging uses a number of database tables to store persistent data. These tables include internal state information of JBoss Messaging, persistent messages and security settings. This section lists tables that hold important data.

JBM_MSG_REF, JBM_MSG
: These tables store persistent messages and their states.

JBM_TX, JBM_TX_EX
: These tables store transaction states.

JBM_USER, JBM_ROLE
: These tables store user and role information.

JBM_POSTOFFICE
: This table holds bindings information.

2.1.1.2. JBoss Messaging configuration files

Most configuration files are stored in **`$JBOSS_HOME/server/$PROFILE/deploy/messaging`**, assuming your JBoss Messaging server profile is **`messaging`**. Applications can choose other locations in which to deploy some configuration files.

You will need to back up and migrate the following configuration files to HornetQ:

Connection Factory service configuration files
: Contain JMS connection factories deployed with the JBoss Messaging server.

Destination service configuration files
: Contain JMS queues and topics deployed with the JBoss Messaging server.

Bridge service configuration files
: Contain bridge services deployed with the JBoss Messaging server.

Other configuration files, such as **`messaging-service.xml`** and the database persistence configuration file, are JBoss Messaging MBeans configurations. The HornetQ implementation consists only of Plain Old Java Objects (POJOs), so these configuration files are not migration targets.

JBoss Messaging relies on the JBoss Remoting and JGroups services in order to work. The configuration files for these services contain settings specific to applications. HornetQ's transport layer and cluster design differ from that in JBoss Messaging. You will need to map the parameters in the service configuration files to their HornetQ equivalents, if any.

2.2. Application Code

If you are using standard JMS in your application there is no need to change your source code.

If you are using JBoss Messaging proprietary features, such as ordering groups, you will need to adapt them to HornetQ equivalent features. For convenience, the following table lists the JBoss Messaging JMS implementation classes and their corresponding HornetQ equivalents.

Table 2.1. Implementation class mapping between JBoss Messaging and HornetQ

org.jboss.jms.client. Classname	HornetQ Equivalent Classname
`JBossConnectionFactory`	`org.hornetq.jms.client.HornetQConnectionFactory`
`JBossConnection`	`org.hornetq.jms.client.HornetQConnection`
`JBossSession`	`org.hornetq.jms.client.HornetQSession`
`JBossMessageProducer`	`org.hornetq.jms.client.HornetQMessageProducer`
`JBossMessageConsumer`	`org.hornetq.jms.client.HornetQMessageConsumer`

Note

Unless you have used JBoss Messaging-specific APIs, it is not necessary to explicitly case your JMS objects to specific implementations. You can just use the standard JMS APIs wherever possible.

2.3. Installing HornetQ

See the *Installation Guide* for up-to-date details on installing HornetQ with this distribution of JBoss Enterprise Application Platform.

2.4. Server Configuration Migration

HornetQ configuration differs significantly from JBoss Messaging, so it is not possible to provide a one-to-one mapping between the two. The following table lists JBoss Messaging server attributes and their equivalents (where possible) in HornetQ.

Unless explicitly indicated, attributes in the HornetQ Server Attributes column are configured in **`hornetq-configuration.xml`**.

Table 2.2. Server attribute mapping between JBoss Messaging and HornetQ

JBoss Messaging Server Attributes (Server Peer MBean)	Equivalent HornetQ Server Attributes
`ServerPeerID`	N/A - HornetQ does not require a specified server ID
`DefaultQueueJNDIContext`, **`DefaultTopicJNDIContext`**	N/A
`PostOffice`	N/A
`DefaultDLQ`	N/A - HornetQ defines dead letter addresses at the core level. There is no default dead letter address for an address unless you specify one.
`DefaultMaxDeliveryAttempts`	N/A - In HornetQ, the default is always **`10`**.
`DefaultExpiryQueue`	N/A - HornetQ defines expiry addresses at the core level. There is no default expiry address for an address unless you specify one.
`DefaultRedeliveryDelay`	N/A - HornetQ's default redelivery delay is always **`0`** (no delay).
`MessageCounterSamplePeriod`	**`message-counter-sample-period`**
`FailoverStartTimeout`	N/A
`FailoverCompleteTimeout`	N/A
`DefaultMessageCounterHistoryDayLimit`	N/A
`ClusterPullConnectionFactory`	N/A
`DefaultPreserveOrdering`	N/A
`RecoverDeliveriesTimeout`	N/A
`EnableMessageCounters`	**`Message-counter-enabled`**
`SuckerPassword`	**`Cluster-password`**
`SuckerConnectionRetryTimes`	**`bridges.reconnect-attempts`**
`SuckerConnectionRetryInterval`	bridges.reconnect-interval
`StrictTCK`	N/A
`Destinations`, **`MessageCounters`**, **`MessageStatistics`**	N/A - These are part of HornetQ's management functions. Refer to the appropriate chapter for details.
`SupportsFailover`	N/A
`PersistenceManager`	N/A - HornetQ uses its built-in high-performance journal as its persistence utility.
`JMSUserManager`	N/A
`SecurityStore`	N/A - The security manager is configured in **`hornetq-beans.xml`** or **`hornetq-jboss-beans.xml`**.

2.5. Migrating JMS-administered Objects and Bridges

HornetQ creates and deploys JMS connection factories, destinations and bridges differently to JBoss Messaging.

In JBoss Messaging, JMS objects and bridges are configured as MBean services within the application server. In HornetQ, these are implemented as POJOs. To migrate the configuration of these objects and bridges from JBoss Messaging to HornetQ, you need to know how parameters map from JBoss Messaging to HornetQ. The following table maps these parameters. Unless otherwise described, all HornetQ attributes are specified in **`hornetq-jms.xml`**.

Table 2.3. JMS Connection Factory Configuration Mappings

JBoss Messaging ConnectionFactory Attributes	HornetQ JMS ConnectionFactory Attributes
`ClientID`	**`connection-factory.client-id`**
`JNDIBindings`	**`connection-factory.entries`**
`PrefetchSize`	**`connection-factory.consumer-window-size`**
`SlowConsumers`	N/A - equivalent to **`consumer-window-size=0`**
`StrictTck`	N/A
`SendAcksAsync`	**`connection-factory.block-on-acknowledge`**
`DefaultTempQueueFullSize`, **`DefaultTempQueuePageSize`**, **`DefaultTempQueueDownCacheSize`**	N/A
`DupsOKBatchSize`	**`connection-factory.dups-ok-batch-size`**
`SupportsLoadBalancing`	N/A
`SupportsFailover`	N/A
`DisableRemotingChecks`	N/A
`LoadBalancingFactory`	**`connection-factory.connection-load-balancing-policy-class-name`**
`Connector`	**`connection-factory.connectors`**
`EnableOrderingGroup`, **`DefaultOrderingGroup`**	N/A

The table below shows how JBoss Messaging Queue attributes map to HornetQ JMS Queue attributes. Unless otherwise specified, these attributes are defined in **`hornetq-configuration.xml`**.

Table 2.4. JMS Queue Configuration Mappings

JBoss Messaging Queue Attributes	HornetQ JMS Queue Attributes
`Name`	**`queue.name`** - defined in **`hornetq-jms.xml`**
`JNDIName`	**`queue.entry`** - defined in **`hornetq-jms.xml`**
`DLQ`	**`address-settings.dead-letter-address`**
`ExpiryQueue`	**`address-settings.expiry-address`**
`RedeliveryDelay`	**`address-settings.redelivery-delay`**

JBoss Messaging Queue Attributes	HornetQ JMS Queue Attributes
MaxDeliveryAttempts	**address-settings.max-delivery-attempts**
SecurityConfig	**security-settings**
FullSize	**address-settings.max-size-bytes** - HornetQ paging attributes do not exactly match JBoss Messaging paging attributes. Refer to the appropriate chapter for details.
PageSize	**address-settings.page-size-bytes** - HornetQ paging attributes do not exactly match JBoss Messaging paging attributes. Refer to the appropriate chapter for details.
DownCacheSize	N/A
CreatedProgrammatically	Refer to **org.hornetq.api.jms.management.JMSQueueControl** to retrieve this attribute.
MessageCount	Refer to **org.hornetq.api.jms.management.JMSQueueControl** to retrieve this attribute.
ScheduledMessageCount	Refer to **org.hornetq.api.jms.management.JMSQueueControl** to retrieve this attribute.
MessageCounter	Refer to **org.hornetq.api.jms.management.JMSQueueControl** to retrieve this attribute.
MessageCounterStatistics	Refer to **org.hornetq.api.jms.management.JMSQueueControl** to retrieve this attribute.
ConsumerCount	Refer to **org.hornetq.api.jms.management.JMSQueueControl** to retrieve this attribute.
DropOldMessageOnRedeploy	N/A
MaxSize	N/A
Clustered	N/A

The table below shows how JBoss Messaging Topic attributes map to HornetQ JMS Topic attributes. Unless otherwise specified, these attributes are defined in **hornetq-configuration.xml**.

Table 2.5. JMS Topic Configuration Mappings

JBoss Messaging Topic Attributes	HornetQ JMS Topic Attributes
Name	**topic.name** - defined in **hornetq-jms.xml**
JNDIName	**topic.entry** - defined in **hornetq-jms.xml**
DLQ	**address-settings.dead-letter-address**
ExpiryQueue	**address-settings.expiry-address**
RedeliveryDelay	**address-settings.redelivery-delay**
MaxDeliveryAttempts	**address-settings.max-delivery-attempts**
SecurityConfig	**security-settings**

JBoss Messaging Topic Attributes	HornetQ JMS Topic Attributes
FullSize	**address-settings.max-size-bytes** - HornetQ paging attributes do not exactly match JBoss Messaging paging attributes. Refer to the appropriate chapter for details.
PageSize	**address-settings.page-size-bytes** - HornetQ paging attributes do not exactly match JBoss Messaging paging attributes. Refer to the appropriate chapter for details.
DownCacheSize	N/A
CreatedProgrammatically	Refer to **org.hornetq.api.jms.management.TopicControl** to retrieve this attribute.
MessageCounterHistoryDayLimit	Refer to **org.hornetq.api.jms.management.TopicControl** to retrieve this attribute.
MessageCounters	Refer to **org.hornetq.api.jms.management.TopicControl** to retrieve this attribute.
AllMessageCount	Refer to **org.hornetq.api.jms.management.TopicControl** to retrieve this attribute.
DurableMessageCount	Refer to **org.hornetq.api.jms.management.TopicControl** to retrieve this attribute.
NonDurableMessageCount	Refer to **org.hornetq.api.jms.management.TopicControl** to retrieve this attribute.
AllSubscriptionsCount	Refer to **org.hornetq.api.jms.management.TopicControl** to retrieve this attribute.
DurableSubscriptionsCount	Refer to **org.hornetq.api.jms.management.TopicControl** to retrieve this attribute.
NonDurableSubscriptionsCount	Refer to **org.hornetq.api.jms.management.TopicControl** to retrieve this attribute.
MaxSize	N/A
Clustered	N/A
DropOldMessageOnRedeploy	N/A

The table below shows how JBoss Messaging Bridge attributes map to HornetQ JMS Bridge attributes. HornetQ's JMS Bridge attributes are defined in its Bean configuration files. Refer to *Chapter 33, The JMS Bridge* for further details.

Table 2.6. JMS Bridge Configuration Mappings

JBoss Messaging Topic Attributes	HornetQ JMS Topic Attributes
SourceProviderLoader	**SourceCFF**

JBoss Messaging Topic Attributes	HornetQ JMS Topic Attributes
TargetProviderLoader	**TargetCFF**
SourceDestinationLookup	**SourceDestinationFactory**
TargetDestinationLookup	**TargetDestinationFactory**
SourceUsername	Source user name parameter
SourcePassword	Source user password parameter
TargetUsername	Target user name parameter
TargetPassword	Target password parameter
QualityOfServiceMode	Quality of Service parameter
Selector	Selector parameter
MaxBatchSize	**Max batch size parameter**
MaxBatchTime	**Max batch time parameter**
SubName	**Subscription name parameter**
ClientID	**Client ID parameter**
FailureRetryInterval	**Failure retry interval parameter**
MaxRetries	**Max retry times parameter**
AddMessageIDInHeader	**Add Message ID in Header parameter**

2.6. Other Configuration in JBoss Messaging

There are two kinds of configurations for JBoss Messaging dependent services, one for JBoss Remoting and the other for JGroups. Details of this configuration is outside the scope of this document. Consult the JBoss Remoting and JGroups documentation for these details.

HornetQ has its own pluggable transportation architecture and clustering implementation, and currently uses Netty as its transport.

2.7. Migrating Existing Messages

Once you have migrated all JMS destinations to HornetQ, you can migrate existing messages. We recommend using the JMS Bridge to move existing messages from JBoss Messaging to HornetQ. We recommend any prepared transactions be completed with JBoss Messaging.

2.8. Applications that use management APIs

JBoss Messaging exposes its management API through MBean interfaces. HornetQ includes a number of different management APIs (see *Chapter 30, Management* for details).

While JBoss Messaging management APIs are accessed via JMX, HornetQ provides access to its management APIs through:

JMX
: JMX is the standard method of managing Java applications.

Core API
: Management operations are sent to the HornetQ server using Core messages.

JMS API
: Management operations are sent to the HornetQ server using JMS messages

The same functionality can be achieved through all three management methods.

The following table lists the JBoss Messaging management Objects alongside their HornetQ JMX counterparts. For the other management APIs available in HornetQ, see *Chapter 30, Management*.

Table 2.7. JBoss Messaging and HornetQ Management Object API Mappings

org.jboss.jms.server. Class	org.hornetq.api.jms.management. Class
`ServerPeer`	`JMSServerControl`
`connectionfactory.ConnectionFactory`	`ConnectionFactoryControl`
`destination.QueueService`	`JMSQueueControl`
`destination.TopicService`	`TopicControl`

Note

Some JBoss Messaging MBeans have no equivalent in HornetQ; for example, there is no HornetQ equivalent for **`JDBCPersistenceManagerService`** because HornetQ does not require a datasource as JBoss Messaging does.

Getting Started with HornetQ

This chapter gives a brief overview of how to get started with HornetQ. The rest of the book provides more detail about the processes described here. In addition to reading this chapter, it is recommended that you run the examples shipped alongside HornetQ to familiarize yourself with its features.

3.1. Starting The Server

3.1.1. HornetQ and JBoss Enterprise Application Platform

To run HornetQ in JBoss Enterprise Application Platform, you need to create the AS 5 profiles for HornetQ first, then run JBoss Enterprise Application Platform with one these profiles. For example, to run JBoss Enterprise Application Platform with a *non-clustered* HornetQ server, go to **`$JBOSS_HOME/bin`** directory and type:

```
bin$ ./run.sh -c default-with-hornetq

15:18:35,460 INFO  [ServerImpl] Starting JBoss (Microcontainer)...
15:18:35,462 INFO  [ServerImpl] Release ID: JBoss (Microcontainer) [5.1.0.GA (build:
 SVNTag=JBoss_5_1_0_GA date=200905221053)
...
15:19:30,305 INFO  [HornetQServerImpl] HornetQ Server version
2.2.0 (yellowjacket, 111) started
...
15:19:43,601 INFO  [ServerImpl] JBoss (Microcontainer) [5.1.0.GA (build:
 SVNTag=JBoss_5_1_0_GA date=200905221053)]Started in 1m:14s:556ms
```

3.2. Running the Examples

In the examples directory there are two sets of examples, these are:

- JMS Examples - these demonstrate functionality while sending and consuming JMS messages.
- Java EE Examples - these demonstrate application server integration. For example MDBs, EJBs, Servlets, and so on.

JBoss WS CXF Test Suite

The JBoss WS CXF Test Suite can be run against HornetQ on JBoss Enterprise Application Platform by providing the **`-Duse.hornetq=true`** option. This runs the test suite with the HornetQ version of the test archives.

3.2.1. The JMS examples

The JMS Examples all follow the same format. Each example is contained in its own directory which holds the following:

- **`build.xml`**

  This is the ant build file used to run the example

- **`src`** directory

  This contains the source code for the example

- **`server0`** configuration directory

  This contains the configuration files needed to run the server for the example. There may be multiple configuration directories **`server0`**, **`server1`** and so on, for clustered examples.

Each example will start one or more stand-alone servers and stop them after the example has completed.

Important - Standalone Servers Are Not Supported

The standalone servers started by the examples are only suitable for testing purposes. Do not use the standalone servers in a production environment. Standalone servers *are not* supported in enterprise customer production configurations.

Run the queue example as a quick start. For all other examples refer to the main user manual.

First open a Shell or a Command prompt and navigate to the **`examples/jms/queue`** directory.

Type the command **`./build.sh`** (or **`build.bat`** if you are running on Windows), and you should see the following output:

```
Buildfile: build.xml

  run:

  init:

  compile:
       [echo] src.example.dir=/home/andy/projects/hornetq/trunk/examples/jms/queue/src
      [javac] Compiling 5 source files to /home/andy/projects/hornetq/trunk/examples/jms/
queue
  /build/classes

  runExample:
       [java] 10:41:04,149 INFO  @main [JMSExample] hornetq.example.runServer is true
       [java] 10:41:04,149 INFO  @main [JMSExample] starting server with config 'server0'
   logServerOutput true
       [java] 10:41:04,149 INFO  @main [JMSExample] and vm args: -Xms512M,-Xmx512M,-XX:+U
  seParallelGC,-XX:+AggressiveOpts,-XX:+UseFastAccessorMethods,-Djava.util.logging.config.
  file=/home/andy/projects/hornetq/trunk/examples/jms/common/../../../src/config/stand-alone/
no
  n-clustered/logging.properties
       [java] org.hornetq.jms.example.SpawnedJMSServer out:10:41:08,437 INFO  @main [Journal
  StorageManager] AIO journal selected
       [java] org.hornetq.jms.example.SpawnedJMSServer out:10:41:08,437 WARN  @main [Journal
  StorageManager] AIO wasn't located on this platform, it will fall back to using pure Jav
  a NIO. If your platform is Linux, install LibAIO to enable the AIO journal
       [java] org.hornetq.jms.example.SpawnedJMSServer out:10:41:08,437 WARN  @main [Securit
  yStoreImpl] It has been detected that the cluster admin password which is used to replic
  ate management operation from one node to the other has not had its password changed fro
  m the installation default. Please see the HornetQ user guide for instructions o
  n how to do this.
       [java] org.hornetq.jms.example.SpawnedJMSServer out:10:41:10,941 INFO  @main
 [HornetQCo
  nnectionFactory] read only is false
       [java] org.hornetq.jms.example.SpawnedJMSServer out:10:41:10,941 INFO  @main
 [HornetQCo
  nnectionFactory] read only is false
       [java] org.hornetq.jms.example.SpawnedJMSServer out:10:41:10,941 INFO  @main
 [HornetQCo
  nnectionFactory] read only is false
```

```
      [java] org.hornetq.jms.example.SpawnedJMSServer out:10:41:10,991 WARN  @main [JMSServ
 erManagerImpl] Binding for java:/ConnectionFactory already exists
      [java] org.hornetq.jms.example.SpawnedJMSServer out:10:41:10,991 WARN  @main [JMSServ
 erManagerImpl] Binding for java:/XAConnectionFactory already exists
      [java] org.hornetq.jms.example.SpawnedJMSServer out:10:41:11,241 INFO  @main [Messagi
 ngServerImpl] HornetQ Server version 2.0.0.BETA5 (buzz-buzz, 107) started
      [java] org.hornetq.jms.example.SpawnedJMSServer out:10:41:11,241 INFO  @main
[HornetQBoot
 strapServer] HornetQ server started
      [java] org.hornetq.jms.example.SpawnedJMSServer out:STARTED::
      [java] 10:41:11,276 INFO  @main [JMSExample] using server0/client-jndi.properties f
 or jndi
      [java] Sent message: This is a text message
      [java] Received message: This is a text message
      [java]
      [java] ####################
      [java] ###    SUCCESS!   ###
      [java] ####################

 BUILD SUCCESSFUL
 Total time: 13 seconds
```

Your first HornetQ example has run correctly.

3.2.2. The Java EE Examples

The Java EE Examples are examples that require a Java EE application server to run. They include MDB, Servlet, EJB examples and so on. For this you will need the JBoss Enterprise Application Platform installed.

Use the MDB example for the purposes of this guide. Before going any further ensure that the JBoss Enterprise Application Platform is running.

Like the other Java EE examples the MDB example is a Java EE Application which first needs deploying in the JBoss Enterprise Application Platform. The first thing to do is set the **JBOSS_HOME** environment property to the location of the JBoss Enterprise Application Platform. In a Linux shell this would be something like:

```
export JBOSS_HOME=/home/jbossas5.1/build/output/jboss-5.1.0.GA
```

Once set, the application can be deployed from inside the **examples/javaee/mdb** directory run the command:

```
./build.sh deploy
```

This will make a copy of the messaging profile in the JBoss Enterprise Application Platform and start the server.

In the shell window you should see the following output:

```
Buildfile: build.xml

  validate-jboss:

  deploy:

  deploy-resources:
      [copy] Copying 1 file to /home/andy/projects/jbossas5.1/build/output/jboss-5.1.0.GA
```

```
       /server/default-with-hornetq/deploy/hornetq.sar
       [copy] Copying 1 file to /home/andy/projects/jbossas5.1/build/output/jboss-5.1.0.GA
       /server/default-with-hornetq/deploy/hornetq.sar

  init:

  compile:

  jar:
       [jar] Building jar: /home/andy/projects/hornetq/trunk/examples/javaee/mdb/build/mdb-
example.jar

  ear:
       [jar] Building jar: /home/andy/projects/hornetq/trunk/examples/javaee/mdb/build/mdb-
example.ear

  deploy-ear:
       [copy] Copying 1 file to /home/andy/projects/jbossas5.1/build/output/jboss-5.1.0.GA

       .......


       [exec] 16:01:04,607 INFO  [JBossASKernel]  Class:javax.jms.MessageListener
       [exec] 16:01:04,607 INFO  [JBossASKernel] Added bean(jboss.j2ee:ear=mdb-example.ear,
       jar=mdb-example.jar,name=MessageMDBExample,service=EJB3) to KernelDeployment of:
       mdb-example.jar
       [exec] 16:01:04,702 INFO  [EJBContainer] STARTED EJB:
 org.hornetq.javaee.example.server
       .MDBExample ejbName: MessageMDBExample
       [exec] 16:01:09,496 INFO  [TomcatDeployment] deploy, ctxPath=/mdb-example
       [exec] 16:01:09,812 WARNING [config] Unable to process deployment descriptor for cont
       ext '/mdb-example'
       [exec] 16:01:09,812 INFO  [config] Initializing Mojarra (1.2_12-b01-FCS) for context
       '/mdb-example'
       [exec] 16:01:09,958 INFO  [Http11Protocol] Starting Coyote HTTP/1.1 on http-127.0.0.1-
       8080
       [exec] 16:01:09,990 INFO  [AjpProtocol] Starting Coyote AJP/1.3 on ajp-127.0.0.1-8009
       [exec] 16:01:10,001 INFO  [ServerImpl] JBoss (Microcontainer) [5.1.0.GA (build: SVNTag
       =JBoss_5_1_0_GA date=200905151005)] Started in 6m:21s:797ms
```

You can now run the example. Do this by running the following command from within the **mdb** directory.

```
./build.sh
```

You should see output similar to the following:

```
Buildfile: build.xml

  run:

  init:

  compile:

  runExample:
       [java] Sent message: This is a text message

  BUILD SUCCESSFUL
  Total time: 7 seconds
```

You may also see some output from the JBoss Enterprise Application Platform. In this instance it is:

```
16:27:54,703 INFO  [STDOUT] message This is a text message received
```

The application can then be undeployed by running the following command from within the **mdb** directory.

```
./build.sh undeploy
```

This should show similar output to the following:

```
Buildfile: build.xml

  validate-jboss:

  undeploy:

  undeploy-misc:
     [delete] Deleting: /home/andy/projects/jbossas5.1/build/output/jboss-5.1.0.GA
     /server/default-with-hornetq/deploy/mdb-example.ear
     [delete] Deleting: /home/andy/projects/jbossas5.1/build/output/jboss-5.1.0.GA
     /server/default-with-hornetq/deploy/hornetq.sar/hornetq-queues.xml
     [delete] Deleting: /home/andy/projects/jbossas5.1/build/output/jboss-5.1.0.GA
     /server/default-with-hornetq/deploy/hornetq.sar/hornetq-jms.xml

  BUILD SUCCESSFUL
  Total time: 1 second
```

You can verify that the application has been undeployed by checking the output from the JBoss Enterprise Application Platform. In this instance it will be something like:

```
16:36:45,277 INFO  [EJBContainer] STOPPED EJB: org.jboss.javaee.example.server.MDBExample
 ejbName: MessageMDBExample
```

Your Java EE example has successfully been run and deployed.

3.2.2.1. Option: Add JBOSS_HOME Environment Variable

The environment variable `JBOSS_HOME` is not required for the operation of HornetQ. However it is sometimes used by other software to determine the location where your JBoss Application Server is installed. A common example of this is the build scripts for sample applications.

The directory to use for `JBOSS_HOME` is the root directory of the JBoss Enterprise Application Platform installation. *Procedure 3.1, “Set JBOSS_HOME on Red Hat Enterprise Linux”* and *Procedure 3.2, “Set JBOSS_HOME on Microsoft Windows”* describe the method for setting JBOSS_HOME for Linux and Microsoft Windows.

Procedure 3.1. Set JBOSS_HOME on Red Hat Enterprise Linux

1. **Open .bash_profile**
   Open **/home/<user>/.bash_profile** using your preferred text editor.

2. **Set the JBOSS_HOME location**
   At the end of the file append the following line. Replace <*path*> with the path to your JBoss Enterprise Application Platform directory.

   ```
   export JBOSS_HOME=<path>
   ```

For example, if your JBoss Enterprise Application platform is located in the **/opt** directory, you would define JBOSS_HOME in the following way:

```
export JBOSS_HOME=/opt/jboss-eap-5.1/jboss-as
```

Procedure 3.2. Set JBOSS_HOME on Microsoft Windows

1. **Open System Properties**
   Open the System Properties dialog by double-clicking on the System icon in the Control Panel.

2. **Open Environment Variables**
   In the System Properties dialog click the Advanced tab, and then click the Environment Variables to open the Environment Variables dialog

3. **Add a New Environment Variable**
   Click the New button to open the Edit User Variable dialog.

4. **Define the JBOSS_HOME environment variable**
   Set the **Variable name:** to JBOSS_HOME

   Set the **Variable value:** to the path of your JBoss Enterprise Application Platform directory

   Click **OK** to confirm the settings.

Messaging Concepts

HornetQ is an asynchronous messaging system; an example of Message Oriented Middleware, which will be referred to as *messaging systems*.

4.1. Messaging Concepts

Messaging systems allow you to loosely couple heterogeneous systems together, whilst typically providing reliability, transactions and many other features.

Unlike systems based on a Remote Procedure Call (RPC) pattern, messaging systems primarily use an asynchronous message passing pattern with no tight relationship between requests and responses. Most messaging systems also support a request-response mode but this is not a primary feature of messaging systems.

Designing systems to be asynchronous from end-to-end allows you to really take advantage of your hardware resources, minimizing the amount of threads blocking on IO operations, and to use your network bandwidth to its full capacity. With an RPC approach you have to wait for a response for each request you make so are limited by the network round trip time, or *latency* of your network. With an asynchronous system you can pipeline flows of messages in different directions, so are limited by the network *bandwidth* not the latency. This typically allows you to create much higher performance applications.

Messaging systems decouple the senders of messages from the consumers of messages. The senders and consumers of messages are completely independent and know nothing of each other. This allows you to create flexible, loosely coupled systems.

Large enterprises often use a messaging system to implement a message bus which loosely couples heterogeneous systems together. Message buses often form the core of an Enterprise Service Bus (ESB). Using a message bus to decouple disparate systems can allow the system to grow and adapt more easily. It also allows more flexibility to add new systems or retire old ones since they don't have brittle dependencies on each other.

4.2. Messaging styles

Messaging systems normally support two main styles of asynchronous messaging: message queue messaging (also known as *point-to-point messaging*) and publish subscribe messaging. They are summarized briefly here:

4.2.1. The Message Queue Pattern

With this type of messaging you send a message to a queue. The message is then typically persisted to provide a guarantee of delivery. Some time later the messaging system delivers the message to a consumer. The consumer processes the message and acknowledges the message when it is done. Once the message is acknowledged it disappears from the queue and is not available to be delivered again. If the system crashes before the messaging server receives an acknowledgment from the consumer, the message will be available to be delivered to a consumer again, upon recovery.

With point-to-point messaging, there can be many consumers in the queue but a particular message will only ever be consumed by one of them. Senders (also known as *producers*) to the queue are completely decoupled from receivers (also known as *consumers*) of the queue; that is, they do not know of each other's existence.

A classic example of point-to-point messaging would be an order queue in a company's book ordering system. Each order is represented as a message which is sent to the order queue. Let's imagine

there are many front end ordering systems which send orders to the order queue. When a message arrives on the queue it is persisted; this ensures that if the server crashes the order is not lost. Let's also imagine there are many consumers on the order queue; each representing an instance of an order processing component - these can be on different physical machines but consuming from the same queue. The messaging system delivers each message to only one of the ordering processing components. Different messages can be processed by different order processors, but a single order is only processed by one order processor - this ensures orders aren't processed twice.

As an order processor receives a message, it fulfills the order, sends order information to the warehouse system and then updates the order database with the order details. Once it's done that, it acknowledges the message to tell the server that the order has been processed and can be forgotten about. Often the send to the warehouse system, update in database and acknowledgment will be completed in a single transaction to ensure ACID properties.

4.2.2. The Publish-Subscribe Pattern

With publish-subscribe messaging, many senders can send messages to an entity on the server, often called a *topic* (it is used this way in the JMS world, for example).

There can be many *subscriptions* on a topic; a subscription is just another word for a consumer of a topic. Each subscription receives a *copy* of *each* message sent to the topic. This differs from the message queue pattern where each message is only consumed by a single consumer.

Subscriptions can optionally be *durable*, which means they retain a copy of each message sent to the topic until the subscriber consumes them - even if the server crashes or is restarted in between. Non-durable subscriptions only last a maximum of the lifetime of the connection that created them.

An example of publish-subscribe messaging would be a news feed. As news articles are created by different editors around the world they are sent to a news feed topic. There are many subscribers around the world who are interested in receiving news items - each one creates a subscription and the messaging system ensures that a copy of each news message is delivered to each subscription.

4.3. Delivery guarantees

A key feature of most messaging systems is *reliable messaging*. With reliable messaging the server gives a guarantee that the message will be delivered only once to each consumer of a queue or each durable subscription of a topic, even in the event of system failure. This is crucial for many businesses; you don't want your orders fulfilled more than once or any of your orders to be lost, for example.

In other cases you may not care about a once only delivery guarantee and are happy to cope with duplicate deliveries or lost messages. An example of this might be transient stock price updates, which are quickly superseded by the next update on the same stock. The messaging system allows you to configure which delivery guarantees you require.

4.4. Transactions

Messaging systems typically support the sending and acknowledgment of multiple messages in a single local transaction. HornetQ also supports the sending and acknowledgment of messages as part of a large global transaction by using the Java mapping of XA; JTA.

4.5. Durability

Messages are either durable or non durable. Durable messages will be persisted in permanent storage and will survive server failure or restart. Non durable messages will not survive server failure

or restart. Examples of durable messages might be orders or trades, where they cannot be lost. An example of a non durable message might be a stock price update which is transitory and doesn't need to survive a restart.

4.6. Messaging APIs and protocols

Many messaging systems provide their own proprietary APIs, which the client can use to communicate with the messaging system.

There are also some standard ways of operating with messaging systems and some emerging standards in this space. Some of these are explored in the next section.

4.6.1. Java Message Service (JMS)

JMS is part of Sun's Java EE specification. It's a Java API that encapsulates both message queue and publish-subscribe messaging patterns. JMS is a lowest common denominator specification. That is, it was created to encapsulate common functionality of the messaging systems that already existed and were available at the time of its creation.

JMS is a very popular API and is implemented by most messaging systems. JMS is only available to clients running Java.

JMS does not define a standard wire format; it only defines a programmatic API so JMS clients and servers from different vendors cannot directly interoperate since each will use the vendor's own internal wire protocol.

HornetQ provides a fully compliant JMS 1.1 API.

4.6.2. System specific APIs

Many systems provide their own programmatic API to interact with the messaging system. The advantage of this is that it allows the full set of system functionality to be exposed to the client application. APIs like JMS are not normally rich enough to expose all the extra features that most messaging systems provide.

HornetQ provides its own core client API for clients to use if they wish to have access to functionality beyond that accessible via the JMS API.

4.6.3. STOMP

Stomp is a very simple text protocol for interoperating with messaging systems. It defines a wire format, so theoretically any Stomp client can work with any messaging system that supports Stomp. Stomp clients are available in many different programming languages.

Please see *Section 45.1, “Stomp”* for using STOMP with HornetQ.

4.7. High Availability

High Availability (HA) means that the system should remain operational after failure of one or more of the servers. The degree of support for HA varies between messaging systems.

HornetQ provides automatic failover; where your sessions are automatically reconnected to the backup server in the event of live server failure.

For more information on HA, please see *Chapter 39, High Availability and Failover*.

4.8. Clusters

Many messaging systems allow you to create groups of messaging servers called *clusters*. Clusters allow the load of sending and consuming messages to be spread over many servers. This allows your system to scale horizontally by adding new servers to the cluster.

Degrees of support for clusters varies between messaging systems, with some systems having fairly basic clusters with the cluster members being hardly aware of each other.

HornetQ provides a very configurable clustering model where messages can be intelligently load balanced between the servers in the cluster, according to the number of consumers on each node, and whether they are ready for messages.

HornetQ also has the ability to automatically redistribute messages between nodes of a cluster to prevent on any particular node.

For full details on clustering, please see *Chapter 38, Clusters*.

4.9. Bridges and routing

Some messaging systems allow isolated clusters or single nodes to be bridged together, typically over unreliable connections like a wide area network (WAN), or the Internet.

A bridge normally consumes messages from a queue on one server and routes messages to another queue on a different server. Bridges cope with unreliable connections, automatically reconnecting when the connections becomes available again.

HornetQ bridges can be configured with filter expressions to only forward certain messages, and transformation can also be hooked in.

HornetQ also allows routing between queues to be configured in server side configuration. This allows complex routing networks to be set up forwarding or copying messages from one destination to another, forming a global network of interconnected brokers.

For more information please see *Chapter 36, Core Bridges* and *Chapter 35, Diverting and Splitting Message Flows*.

Core Architecture

HornetQ core is designed simply as a set of Plain Old Java Objects (POJOs). It is designed for simplicity.

It has also been designed to have as few dependencies on external jars as possible. As a result HornetQ core has only one more jar dependency than the standard JDK classes; **`netty.jar`**. This is because some of the netty buffer classes are used internally.

This allows HornetQ to be instantiated in any dependency injection framework such as JBoss Microcontainer, or Spring.

Each HornetQ server has its own ultra high performance persistent journal, which it uses for messaging and other persistence.

Using a high performance journal allows persistence message performance, which is something not achievable when using a relational database for persistence.

HornetQ clients, potentially on different physical machines interact with the HornetQ server. HornetQ currently provides two APIs for messaging at the client side:

Core client API
: This is a simple intuitive Java API that allows the full set of messaging functionality without some of the complexities of JMS.

JMS client API
: The standard JMS API is available at the client side.

JMS semantics are implemented by a thin JMS facade layer on the client side.

The HornetQ server does not associate with JMS and does not know anything about JMS. It is a protocol agnostic messaging server designed to be used with multiple different protocols.

When a user uses the JMS API on the client side, all JMS interactions are translated into operations on the HornetQ core client API before being transferred over the wire using the HornetQ wire format.

The server always just deals with core API interactions.

A schematic illustrating this relationship is described in *Figure 5.1, “HornetQ Application Interaction Schematic”*.

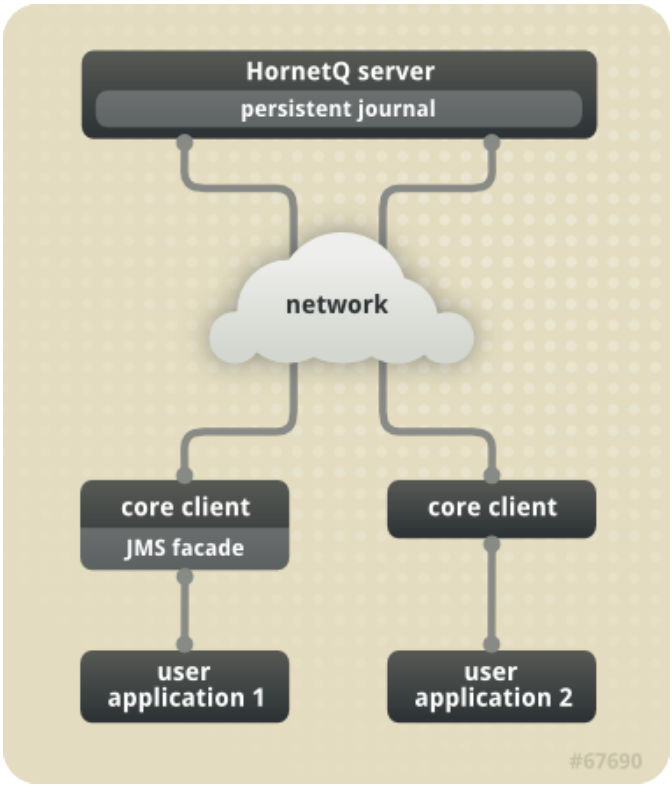


Figure 5.1. HornetQ Application Interaction Schematic

Figure 3.1 shows two user applications interacting with a HornetQ server. User Application 1 is using the JMS API, while User Application 2 is using the core client API directly.

You can see from the diagram that the JMS API is implemented by a thin facade layer on the client side.

Using the Server

This chapter will familiarize you with how to use the HornetQ server.

For the remainder of this chapter, the *HornetQ server* refers to the HornetQ server, in its default configuration with a JMS Service and JNDI service enabled.

6.1. Library Path

If you're using the Asynchronous IO Journal on Linux, you need to specify **`java.library.path`** as a property on your Java options. This is done automatically in the **`run.sh`** script.

If you don't specify **`java.library.path`** at your Java options then the JVM will use the environment variable **`LD_LIBRARY_PATH`**.

6.2. System properties

HornetQ can take a system property on the command line for configuring logging.

For more information on configuring logging, please see *Chapter 43, Logging*.

6.3. Configuration files

The configuration files are stored in a number of locations in the HornetQ directory structure. In all cases, you must change the file in each server profile you want to run because they are not shared between profiles.

Files located in /deploy/hornetq/

hornetq-configuration.xml

This is the main HornetQ configuration file. All the parameters in this file are described in *Chapter 47, Configuration Reference*. Refer to *Section 6.4, "The main configuration file."* for more information on this file.

Note

The property **`file-deployment-enabled`** in the **`hornetq-configuration.xml`** configuration when set to false means that the other configuration files are not loaded. By default, this is set to true.

hornetq-jboss-beans.xml

This is the JBoss Microcontainer beans file which defines what beans the Microcontainer should create and what dependencies to enforce between them.

hornetq-jms.xml

The distribution configuration by default includes a server side JMS service which mainly deploys JMS Queues, Topics and Connection Factories from this file into JNDI. If you're not using JMS, or you don't need to deploy JMS objects on the server side, then you don't need this file. For more information on using JMS, please see *Chapter 7, Using JMS*.

hornetq-queues.xml

This file contains predefined queues, queue settings and security settings. The file is optional; all this configuration can also live in **`hornetq-configuration.xml`**. In fact, the default

configuration sets do not have a **hornetq-queues.xml** file. The purpose of allowing queues to be configured in these files is to allow you to manage your queue configuration over many files instead of being forced to maintain it in a single file. There can be many **hornetq-queues.xml** files on the classpath. All will be loaded if found.

Files located in /conf/props/

hornetq-users.properties

HornetQ ships with a security manager implementation that obtains user credentials from the **hornetq-users.properties** file. This file contains user and password information. For more information on security, refer to *Chapter 31, Security*.

hornetq-roles.properties

This file contains user names defined in **hornetq-users.properties** with the roles they have permission to use. For more information on security, refer to *Chapter 31, Security*.

It is also possible to use system property substitution in all the configuration files in a server profile by replacing a value with the name of a system property. Here is an example of this with a connector configuration:

```
<connector name="netty">
   <factory-class>org.hornetq.core.remoting.impl.netty.NettyConnectorFactory
</factory-class>
   <param key="host" value="${hornetq.remoting.netty.host:localhost}" type="String"/>
   <param key="port"  value="${hornetq.remoting.netty.port:5445}" type="Integer"/>
</connector>
```

Here you can see two values have been replaced with system properties **hornetq.remoting.netty.host** and **hornetq.remoting.netty.port**. These values will be replaced by the value found in the system property if there is one. If not, they default back to localhost or 5445 respectively. It is also possible to not supply a default. That is, **${hornetq.remoting.netty.host}**, however the system property *must* be supplied in that case.

6.4. The main configuration file.

The configuration for the HornetQ core server is contained in **hornetq-configuration.xml**. This is what the **FileConfiguration** bean uses to configure the messaging server.

There are many attributes which you can configure in HornetQ. In most cases the defaults will be sufficient. Every attribute can be defaulted which means a file with a single empty **configuration** element is a valid configuration file. The different configuration will be explained throughout the manual or you can refer to the configuration reference *Chapter 47, Configuration Reference*.

Using JMS

Although HornetQ provides a JMS agnostic messaging API, many users will be more comfortable using JMS.

JMS is a very popular API standard for messaging, and most messaging systems provide a JMS API.

HornetQ also ships with a range of examples, many of which demonstrate JMS API usage. A good place to start would be to play around with the simple JMS queue and topic example, but examples are also provided for many other parts of the JMS API. A full description of the examples is available in *Section 3.2, "Running the Examples"*.

This section will cover the main steps in configuring the server for JMS and creating a simple JMS program. It will also show how to configure and use JNDI, and how to use JMS with HornetQ without using any JNDI.

7.1. A Simple Ordering System

This example uses a single JMS Queue called **`OrderQueue`**, and will have a single **`MessageProducer`** sending an order message to the queue and a single **`MessageConsumer`** consuming the order message from the queue.

The queue will be a **`durable`** queue, that is, it will survive a server restart or crash. The example will also show how to specify the queue in the server JMS configuration so it is created automatically without having to explicitly create it from the client.

7.2. JMS Server Configuration

The file **`hornetq-jms.xml`** on the server classpath contains any JMS queue, topic and ConnectionFactory instances that we wish to create and make available to lookup via the JNDI.

A JMS ConnectionFactory object is used by the client to make connections to the server. It knows the location of the server it is connecting to, as well as many other configuration parameters. In most cases the defaults will be acceptable.

The example will deploy a single JMS queue and a single JMS Connection Factory instance on the server for this example but there are no limits to the number of queues, topics and ConnectionFactory instances you can deploy from the file. Here's the configuration:

```
<configuration xmlns="urn:hornetq" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="urn:hornetq ../schemas/hornetq-jms.xsd ">
   <connection-factory name="ConnectionFactory">
      <connectors>
         <connector-ref connector-name="netty"/>
      </connectors>
      <entries>
         <entry name="ConnectionFactory"/>
      </entries>
   </connection-factory>
   <queue name="OrderQueue">
      <entry name="queues/OrderQueue"/>
   </queue>
</configuration>
```

One ConnectionFactory called **`ConnectionFactory`** is deployed and bound in just one place in JNDI as given by the **`entry`** element. ConnectionFactory instances can be bound in many places in JNDI if it is required.

Note

The JMS connection factory references a **`connector`** called **`netty`**. This is a reference to a connector object deployed in the main core configuration file **`hornetq-configuration.xml`**, which defines the transport and parameters used to actually connect to the server.

7.3. Connection Factory Types

The JMS API doc provides several connection factories for applications. HornetQ JMS users can choose to configure the types for their connection factories. Each connection factory has a **`signature`** attribute and a **`xa`** parameter, the combination of which determines the type of the factory. Attribute **`signature`** has three possible string values, i.e. *generic*, *queue* and *topic*; **`xa`** is a boolean type parameter. The following table gives their configuration values for different connection factory interfaces.

Table 7.1. Configuration for Connection Factory Types

signature	xa	Connection Factory Type
generic (default)	false (default)	javax.jms.ConnectionFactory
generic	true	javax.jms.XAConnectionFactory
queue	false	javax.jms.QueueConnectionFactory
queue	true	javax.jms.XAQueueConnectionFactory
topic	false	javax.jms.TopicConnectionFactory
topic	true	javax.jms.XATopicConnectionFactory

As an example, the following configures an XAQueueConnectionFactory:

```
<configuration xmlns="urn:hornetq"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="urn:hornetq ../schemas/hornetq-jms.xsd ">

    <connection-factory name="ConnectionFactory" signature="queue">
        <xa>true</xa>
        <connectors>
           <connector-ref connector-name="netty"/>
        </connectors>
        <entries>
            <entry name="ConnectionFactory"/>
        </entries>
    </connection-factory>
</configuration>
```

7.4. JNDI configuration

When using JNDI from the client side you need to specify a set of JNDI properties that tell the JNDI client where to locate the JNDI server. These are often specified in a file called **`jndi.properties`** on the client classpath, or you can specify them directly when creating the JNDI initial context. A full JNDI

tutorial is outside the scope of this document, please see the *Sun JNDI tutorial*[1] for more information on how to use JNDI.

For talking to the JBoss JNDI Server, the jndi properties will look something like this:

```
java.naming.factory.initial=org.jnp.interfaces.NamingContextFactory
java.naming.provider.url=jnp://myhost:1099
java.naming.factory.url.pkgs=org.jboss.naming:org.jnp.interfaces
```

Where **myhost** is the hostname or IP address of the JNDI server. 1099 is the port used by the JNDI server and may vary depending on how you have configured your JNDI server.

In the default configuration, JNDI server ports are configured in the file **hornetq-beans.xml** by setting properties on the **JNDIServer** bean:

```
<bean name="JNDIServer" class="org.jnp.server.Main">
    <property name="namingInfo">
        <inject bean="Naming"/>
    </property>
    <property name="port">1099</property>
    <property name="bindAddress">localhost</property>
    <property name="rmiPort">1098</property>
    <property name="rmiBindAddress">localhost</property>
</bean>
```

Note

If you want your JNDI server to be available to non local clients make sure you change it's bind address to something other than **localhost**.

7.5. The code

The code for the example is available below.

The first step is to create a JNDI initial context from which to look up JMS objects.

```
InitialContect ic = new InitialContext();
```

The next step is to look up the connection factory:

```
ConnectionFactory cf = (ConnectionFactory)ic.lookup("/ConnectionFactory");
```

Followed by looking up the Queue:

```
Queue orderQueue = (Queue)ic.lookup("/queues/OrderQueue");
```

Next, create a JMS connection using the connection factory:

```
Connection connection = cf.createConnection();
```

Create a non transacted JMS Session, with AUTO_ACKNOWLEDGE acknowledge mode:

[1] http://java.sun.com/products/jndi/tutorial/TOC.html

```
Session session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE);
```

Create a MessageProducer that will send orders to the queue:

```
MessageProducer producer = session.createProducer(orderQueue);
```

Create a MessageConsumer which will consume orders from the queue:

```
MessageConsumer consumer = session.createConsumer(orderQueue);
```

Make sure to start the connection, or delivery will not occur on it:

```
connection.start();
```

Create a simple TextMessage and send it:

```
TextMessage message = session.createTextMessage("This is an order");
producer.send(message);
```

Consume the message:

```
TextMessage receivedMessage = (TextMessage)consumer.receive();
System.out.println("Got order: " + receivedMessage.getText());
```

For a wide range of working JMS examples please see the examples directory in the distribution.

Warning

Please note that JMS connections, sessions, producers and consumers are *designed to be re-used*.

It's an anti-pattern to create new connections, sessions, producers and consumers for each message you produce or consume. If you do this, your application will perform very poorly. This is discussed further in the section on performance tuning *Chapter 46, Performance Tuning*.

7.6. Directly instantiating JMS Resources without using JNDI

It's a very common usage pattern to look up JMS *Administered Objects* (that's JMS queue, topic and ConnectionFactory instances) from JNDI. However in some cases, a JNDI server is not available, and using JMS is still required, or it is preferable to directly instantiate objects. This is possible with HornetQ, which supports the direct instantiation of JMS queue, topic and ConnectionFactory instances.

For a full working example of direct instantiation please see the JMS examples in *Section 3.2, "Running the Examples"*.

The following is a simple example, which does not use JNDI at all:

Create the JMS ConnectionFactory object via the HornetQJMSClient Utility class. Note you need to provide connection parameters and specify which transport you are using. For more information on connectors please see *Chapter 16, Configuring the Transport*.

```
TransportConfiguration transportConfiguration =
                     new TransportConfiguration(NettyConnectorFactory.class.getName());

ConnectionFactory cf = HornetQJMSClient.createConnectionFactory(transportConfiguration);
```

Also create the JMS Queue object via the HornetQJMSClient Utility class:

```
Queue orderQueue = HornetQJMSClient.createQueue("OrderQueue");
```

Next create a JMS connection using the connection factory:

```
Connection connection = cf.createConnection();
```

Create a non transacted JMS Session, with AUTO_ACKNOWLEDGE acknowledge mode:

```
Session session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE);
```

Create a MessageProducer that will send orders to the queue:

```
MessageProducer producer = session.createProducer(orderQueue);
```

Create a MessageConsumer which will consume orders from the queue:

```
MessageConsumer consumer = session.createConsumer(orderQueue);
```

Make sure you start the connection, or delivery will not occur on it:

```
connection.start();
```

Create a simple TextMessage and send it:

```
TextMessage message = session.createTextMessage("This is an order");
producer.send(message);
```

And we consume the message:

```
TextMessage receivedMessage = (TextMessage)consumer.receive();
System.out.println("Got order: " + receivedMessage.getText());
```

7.7. Setting The Client ID

This represents the client id for a JMS client and is needed for creating durable subscriptions. It is possible to configure this on the connection factory and can be set via the **`client-id`** element. Any connection created by this connection factory will have this set as its client id.

7.8. Setting The Batch Size for DUPS_OK

When the JMS acknowledge mode is set to **`DUPS_OK`** it is possible to configure the consumer so that it sends acknowledgments in batches rather that one at a time, saving valuable bandwidth. This can be configured via the connection factory via the **`dups-ok-batch-size`** element and is set in bytes. The default is 1024 * 1024 bytes = 1 MiB (Mebibyte).

7.9. Setting The Transaction Batch Size

When receiving messages in a transaction it is possible to configure the consumer to send acknowledgments in batches rather than individually saving valuable bandwidth. This can be configured on the connection factory via the **`transaction-batch-size`** element and is set in bytes. The default is 1024 * 1024.

Using Core

HornetQ core is a completely JMS-agnostic messaging system with its own non-JMS API. This is called the *core API*.

You can use the core API directly if you do not want to use JMS. The core API provides all the functionality of JMS but without much of the complexity. It also provides features that are not available using JMS.

8.1. Core Messaging Concepts

Some of the core messaging concepts are similar to JMS concepts, but there are still differences between them. In general, the core messaging API is simpler than the JMS API, since distinctions between queues, topics and subscriptions are removed. Each of these major core messaging concepts will be discussed in turn.

8.1.1. Message

- A message is the unit of data that is sent between clients and servers.
- A message has a body which is a buffer containing convenient methods for reading and writing data into it.
- A message has a set of properties which are key-value pairs. Each property key is a string and property values can be of type integer, long, short, byte, byte[], String, double, float or boolean.
- A message has an *address* it is being sent to. When the message arrives on the server it is routed to any queues that are bound to the address - if the queues are bound with any filter, the message will only be routed to that queue if the filter matches. An address may have many queues bound to it or none. There may also be entities other than queues, like *diverts* bound to addresses.
- Messages can be either durable or non durable. Durable messages in a durable queue will survive a server crash or restart. Non durable messages will not survive a server crash or restart.
- Messages can be specified with a priority value between 0 and 9. 0 represents the lowest priority and 9 represents the highest. HornetQ will attempt to deliver higher priority messages before lower priority ones.
- Messages can be specified with an optional expiry time. HornetQ will not deliver messages after its expiry time has been exceeded.
- Messages also have an optional time stamp which represents the time the message was sent.
- HornetQ also supports the sending or consuming of very large messages - much larger than can fit in available RAM at any one time.

8.1.2. Address

A server maintains a mapping between an address and a set of queues. Zero or more queues can be bound to a single address. Each queue can be bound with an optional message filter. When a message is routed, it is routed to the set of queues bound to the message's address. If any of the queues are bound with a filter expression, then the message will only be routed to the subset of bound queues which match that filter expression.

Other entities, such as *diverts* can also be bound to an address and messages will also be routed there.

Note

In core, there is no concept of a topic; topic is a JMS only term. Instead, in core, we just deal with *addresses* and *queues*.

For example, a JMS topic would be implemented by a single address to which many queues are bound. Each queue represents a subscription of the topic. A JMS queue would be implemented as a single address to which one queue is bound; that queue represents the JMS queue.

8.1.3. Queue

Queues can be durable, which means the messages they contain survive a server crash or restart, as long as the messages in them are durable. Non-durable queues do not survive a server restart or crash even if the messages they contain are durable.

Queues can also be temporary, meaning they are automatically deleted when the client connection is closed, if they are not explicitly deleted before that.

Queues can be bound with an optional filter expression. If a filter expression is supplied then the server will only route messages that match that filter expression to any queues bound to the address.

Many queues can be bound to a single address. A particular queue is only bound to a maximum of one address.

8.1.4. ClientSessionFactory

Clients use **`ClientSessionFactory`** instances to create **`ClientSession`** instances. **`ClientSessionFactory`** instances know how to connect to the server to create sessions, and are configurable with many settings.

`ClientSessionFactory` instances are created using the **`HornetQClient`** factory class.

8.1.5. ClientSession

A client uses a ClientSession for consuming and producing messages and for grouping them in transactions. ClientSession instances can support both transactional and non transactional semantics and also provide an **`XAResource`** interface so messaging operations can be performed as part of a JTA transaction.

ClientSession instances group ClientConsumers and ClientProducers.

ClientSession instances can be registered with an optional **`SendAcknowledgementHandler`**. This allows your client code to be notified asynchronously when sent messages have successfully reached the server. This feature allows you to have full guarantees that sent messages have reached the server without having to block on each message sent until a response is received. Blocking on each messages sent is costly since it requires a network round trip for each message sent. By not blocking and receiving send acknowledgments asynchronously you can create true end-to-end asynchronous systems which is not possible using the standard JMS API. For more information on this advanced feature please see the section *Chapter 20, Guarantees of sends and commits*.

8.1.6. ClientConsumer

Clients use **`ClientConsumer`** instances to consume messages from a queue. Core Messaging supports both synchronous and asynchronous message consumption semantics. **`ClientConsumer`**

instances can be configured with an optional filter expression and will only consume messages which match that expression.

8.1.7. ClientProducer

Clients create **`ClientProducer`** instances on **`ClientSession`** instances so they can send messages. ClientProducer instances can specify an address to which all sent messages are routed, or they can have no specified address, and the address is specified at send time for the message.

Warning

Please note that ClientSession, ClientProducer and ClientConsumer instances are *designed to be re-used*.

It's an anti-pattern to create new ClientSession, ClientProducer and ClientConsumer instances for each message you produce or consume. If you do this, your application will not perform to standard. This is discussed further in the section on performance tuning *Chapter 46, Performance Tuning*.

8.2. A simple example of using Core

Here's a very simple program using the core messaging API to send and receive a message:

```
ClientSessionFactory factory =  HornetQClient.createClientSessionFactory(
                                        new TransportConfiguration(
                                           InVMConnectorFactory.class.getName()));

ClientSession session = factory.createSession();

session.createQueue("example", "example", true);

ClientProducer producer = session.createProducer("example");

ClientMessage message = session.createMessage(true);

message.getBodyBuffer().writeString("Hello");

producer.send(message);

session.start();

ClientConsumer consumer = session.createConsumer("example");

ClientMessage msgReceived = consumer.receive();

System.out.println("message = " + msgReceived.getBodyBuffer().readString());

session.close();
```

Mapping JMS Concepts to the Core API

This chapter describes how JMS destinations are mapped to HornetQ addresses.

HornetQ core is JMS-agnostic. It does not have any concept of a JMS topic. A JMS topic is implemented in core as an address (the topic name) with zero or more queues bound to it. Each queue bound to that address represents a topic subscription. Likewise, a JMS Queue is implemented as an address (the JMS Queue name) with one single queue bound to it which represents the JMS Queue.

By convention, all JMS Queues map to core queues where the core queue name has the string **`jms.queue.`** prepended to it. For example, the JMS Queue with the name "orders.europe" would map to the core queue with the name "jms.queue.orders.europe". The address at which the core queue is bound is also given by the core queue name.

For JMS topics the address at which the queues that represent the subscriptions are bound is given by prepending the string "jms.topic." to the name of the JMS Topic. For example the JMS Topic with name "news.europe" would map to the core address "jms.topic.news.europe"

In other words if you send a JMS message to a JMS queue with name "orders.europe" it will get routed on the server to any core queues bound to the address "jms.queue.orders.europe". If you send a JMS message to a JMS topic with name "news.europe" it will get routed on the server to any core queues bound to the address "jms.topic.news.europe".

If you want to configure settings for a JMS Queue with the name "orders.europe", you need to configure the corresponding core queue "jms.queue.orders.europe":

```
<!-- expired messages in JMS Queue "orders.europe"
    will be sent to the JMS Queue "expiry.europe" -->
<address-setting match="jms.queue.orders.europe">
   <expiry-address>jms.queue.expiry.europe</expiry-address>
   ...
</address-setting>
```

The Client Classpath

HornetQ requires several jars on the *Client Classpath* depending on whether the client uses HornetQ Core API, JMS, or JNDI.

Warning

All the jars mentioned here can be found in the ***JBOSS_HOME*/client** directory of the HornetQ distribution. Be sure you only use the jars from the correct version of the release, you *must not* mix and match versions of jars from different HornetQ versions. Mixing and matching different jar versions may cause subtle errors and failures to occur.

10.1. HornetQ Core Client

If you are using just a pure HornetQ Core client (i.e. no JMS) then you need **hornetq-core-client.jar** and **netty.jar** on your client classpath.

If the client runs inside a *Java 5 virtual machine*, use instead **hornetq-core-client-java5.jar**.

10.2. JMS Client

If you are using JMS on the client side, then you will also need to include **hornetq-jms-client.jar** and **jboss-jms-api.jar**.

If the client runs inside a *Java 5 virtual machine*, include instead **hornetq-jms-client-java5.jar**.

Note

jboss-jms-api.jar just contains Java EE API interface classes needed for the **javax.jms.*** classes. If you already have a jar with these interface classes on your classpath, you will not need it.

10.3. JMS Client with JNDI

If you are looking up JMS resources from the JNDI server, you will also need the jar **jnp-client.jar** jar on your client classpath as well as any other jars mentioned previously.

Examples

The HornetQ distribution comes with over 70 run out-of-the-box examples demonstrating many of the features.

The examples are available in the distribution, in the **`examples`** directory. Examples are split into JMS and core examples. JMS examples show how a particular feature can be used by a normal JMS client. Core examples show how the equivalent feature can be used by a core messaging client.

A set of Java EE examples are also provided which need the JBoss Enterprise Application Platform installed to be able to run.

11.1. JMS Examples

To run a JMS example, **`cd`** into the appropriate example directory and type **`./build.sh`** (or **`build.bat`** if you are on Windows).

The following is a list of the examples with a brief description.

11.1.1. Application-Layer Failover

HornetQ also supports Application-Layer failover, which is useful in the case that replication is not enabled on the server side.

With Application-Layer failover, it's up to the application to register a JMS **`ExceptionListener`** with HornetQ which will be called by HornetQ in the event that connection failure is detected.

The code in the **`ExceptionListener`** then recreates the JMS connection, session, and so on, on another node and the application can continue.

Application-layer failover is an alternative approach to High Availability (HA). Application-layer failover differs from automatic failover in that some client side coding is required in order to implement this. Also, with Application-layer failover, since the old session object dies and a new one is created, any uncommitted work in the old session will be lost, and any unacknowledged messages might be redelivered.

11.1.2. Core Bridge Example

The **`bridge`** example demonstrates a core bridge deployed on one server, which consumes messages from a local queue and forwards them to an address on a second server.

Core bridges are used to create message flows between any two HornetQ servers which are remotely separated. Core bridges are resilient and will cope with temporary connection failure allowing them to be an ideal choice for forwarding over unreliable connections, e.g. a WAN.

11.1.3. Browser

The **`browser`** example shows you how to use a JMS **`QueueBrowser`** with HornetQ.

Queues are a standard part of JMS, please consult the JMS 1.1 specification for full details.

A **`QueueBrowser`** is used to look at messages on the queue without removing them. It can scan the entire content of a queue or only messages matching a message selector.

11.1.4. Client Kickoff

The **`client-kickoff`** example shows how to terminate client connections given an IP address using the JMX management API.

11.1.5. Client-Side Load-Balancing

The **`client-side-load-balancing`** example demonstrates how sessions created from a single JMS **`Connection`** can be created to different nodes of the cluster. In other words it demonstrates how HornetQ does client-side load-balancing of sessions across the cluster.

11.1.6. Clustered Grouping

This is similar to the message grouping example except that it demonstrates it working over a cluster. Messages sent to different nodes with the same group id will be sent to the same node and the same consumer.

11.1.7. Clustered Queue

The **`clustered-queue`** example demonstrates a JMS queue deployed on two different nodes. The two nodes are configured to form a cluster. We then create a consumer for the queue on each node, and we create a producer on only one of the nodes. We then send some messages via the producer, and we verify that both consumers receive the sent messages in a round-robin fashion.

11.1.8. Clustered Standalone

Important - Standalone Servers Are Not Supported

The standalone servers started by the examples are only suitable for testing purposes. Do not use the standalone servers in a production environment. Standalone servers *are not* supported in enterprise customer production configurations.

The **`clustered-standalone`** example demonstrates how to configure and starts three cluster nodes on the same machine to form a cluster. A subscriber for a JMS topic is created on each node, and we create a producer on only one of the nodes. We then send some messages via the producer, and we verify that the 3 subscribers receive all the sent messages.

11.1.9. Clustered Topic

The **`clustered-topic`** example demonstrates a JMS topic deployed on two different nodes. The two nodes are configured to form a cluster. We then create a subscriber on the topic on each node, and we create a producer on only one of the nodes. We then send some messages via the producer, and we verify that both subscribers receive all the sent messages.

11.1.10. Message Consumer Rate Limiting

With HornetQ you can specify a maximum consume rate at which a JMS MessageConsumer will consume messages. This can be specified when creating or deploying the connection factory.

If this value is specified then HornetQ will ensure that messages are never consumed at a rate higher than the specified rate. This is a form of consumer throttling.

11.1.11. Dead Letter

The **`dead-letter`** example shows you how to define and deal with dead letter messages. Messages can be delivered unsuccessfully (e.g. if the transacted session used to consume them is rolled back).

Such a message goes back to the JMS destination ready to be redelivered. However, this means it is possible for a message to be delivered again and again without any success and remain in the destination, clogging the system.

To prevent this, messaging systems define dead letter messages: after a specified unsuccessful delivery attempts, the message is removed from the destination and put instead in a dead letter destination where they can be consumed for further investigation.

11.1.12. Delayed Redelivery

The **`delayed-redelivery`** example demonstrates how HornetQ can be configured to provide a delayed redelivery in the case a message needs to be redelivered.

Delaying redelivery can often be useful in the case that clients regularly fail or roll-back. Without a delayed redelivery, the system can get into a "thrashing" state, with delivery being attempted, the client rolling back, and delivery being re-attempted in quick succession, using up valuable CPU and network resources.

11.1.13. Divert

HornetQ diverts allow messages to be transparently "diverted" or copied from one address to another with just some simple configuration defined on the server side.

11.1.14. Durable Subscription

The **`durable-subscription`** example shows you how to use a durable subscription with HornetQ. Durable subscriptions are a standard part of JMS, please consult the JMS 1.1 specification for full details.

Unlike non-durable subscriptions, the key function of durable subscriptions is that the messages contained in them persist longer than the lifetime of the subscriber - i.e. they will accumulate messages sent to the topic even if there is no active subscriber on them. They will also survive server restarts or crashes. Note that for the messages to be persisted, the messages sent to them must be marked as durable messages.

11.1.15. Embedded

The **`embedded`** example shows how to embed the HornetQ server within your own code.

11.1.16. HTTP Transport

The **`http-transport`** example shows you how to configure HornetQ to use the HTTP protocol as its transport layer.

11.1.17. Instantiate JMS Objects Directly

Usually, JMS Objects such as **`ConnectionFactory`**, **`Queue`** and **`Topic`** instances are looked up from JNDI before being used by the client code. This objects are called "administered objects" in JMS terminology.

However, in some cases a JNDI server may not be available or desired. To come to the rescue HornetQ also supports the direct instantiation of these administered objects on the client side so you don't have to use JNDI for JMS.

11.1.18. Interceptor

HornetQ allows an application to use an interceptor to hook into the messaging system. Interceptors allow you to handle various message events in HornetQ.

11.1.19. JAAS

The **`jaas`** example shows you how to configure HornetQ to use JAAS for security. HornetQ can leverage JAAS to delegate user authentication and authorization to existing security infrastructure.

11.1.20. JMX Management

The **`jmx`** example shows how to manage HornetQ using JMX.

11.1.21. Large Message

The **`large-message`** example shows you how to send and receive very large messages with HornetQ. HornetQ supports the sending and receiving of huge messages, much larger than can fit in available RAM on the client or server. Effectively the only limit to message size is the amount of disk space you have on the server.

Large messages are persisted on the server so they can survive a server restart. In other words HornetQ doesn't just do a simple socket stream from the sender to the consumer.

11.1.22. Last-Value Queue

The **`last-value-queue`** example shows you how to define and deal with last-value queues. Last-value queues are special queues which discard any messages when a newer message with the same value for a well-defined last-value property is put in the queue. In other words, a last-value queue only retains the last value.

A typical example for last-value queue is for stock prices, where you are only interested by the latest price for a particular stock.

11.1.23. Load Balanced Clustered Queue

The **`clustered-queue`** example demonstrates a JMS queue deployed on two different nodes. The two nodes are configured to form a cluster.

We then create a consumer on the queue on each node, and we create a producer on only one of the nodes. We then send some messages via the producer, and we verify that both consumers receive the sent messages in a round-robin fashion.

In other words, HornetQ load balances the sent messages across all consumers on the cluster

11.1.24. Management

The **`management`** example shows how to manage HornetQ using JMS Messages to invoke management operations on the server.

11.1.25. Management Notification

The **`management-notification`** example shows how to receive management notifications from HornetQ using JMS messages. HornetQ servers emit management notifications when events of interest occur (consumers are created or closed, addresses are created or deleted, security authentication fails, etc.).

11.1.26. Message Counter

The **`message-counters`** example shows you how to use message counters to obtain message information for a JMS queue.

11.1.27. Message Expiration

The **`expiry`** example shows you how to define and deal with message expiration. Messages can be retained in the messaging system for a limited period of time before being removed. JMS specification states that clients should not receive messages that have been expired (but it does not guarantee this will not happen).

HornetQ can assign an expiry address to a given queue so that when messages are expired, they are removed from the queue and sent to the expiry address. These "expired" messages can later be consumed from the expiry address for further inspection.

11.1.28. Message Group

The **`message-group`** example shows you how to configure and use message groups with HornetQ. Message groups allow you to pin messages so they are only consumed by a single consumer. Message groups are sets of messages that has the following characteristics:

- Messages in a message group share the same group id, i.e. they have same JMSXGroupID string property values
- The consumer that receives the first message of a group will receive all the messages that belongs to the group

11.1.29. Message Group

The **`message-group2`** example shows you how to configure and use message groups with HornetQ via a connection factory.

11.1.30. Message Priority

Message Priority can be used to influence the delivery order for messages.

It can be retrieved by the message's standard header field 'JMSPriority' as defined in JMS specification version 1.1.

The value is of type integer, ranging from 0 (the lowest) to 9 (the highest). When messages are being delivered, their priorities will effect their order of delivery. Messages of higher priorities will likely be delivered before those of lower priorities.

Messages of equal priorities are delivered in the natural order of their arrival at their destinations. Please consult the JMS 1.1 specification for full details.

11.1.31. No Consumer Buffering

By default, HornetQ consumers buffer messages from the server in a client side buffer before you actually receive them on the client side. This improves performance since otherwise every time you called receive() or had processed the last message in a **`MessageListener onMessage()`** method, the HornetQ client would have to go the server to request the next message, which would then get sent to the client side, if one was available.

This would involve a network round trip for every message and reduce performance. Therefore, by default, HornetQ pre-fetches messages into a buffer on each consumer.

In some case buffering is not desirable, and HornetQ allows it to be switched off. This example demonstrates that.

11.1.32. Non-Transaction Failover With Server Data Replication

The **`non-transaction-failover`** example demonstrates two servers coupled as a live-backup pair for high availability (HA), and a client using a *non-transacted* JMS session failing over from live to backup when the live server is crashed.

HornetQ implements failover of client connections between live and backup servers. This is implemented by the replication of state between live and backup nodes. When replication is configured and a live node crashes, the client connections can carry and continue to send and consume messages. When non-transacted sessions are used, once and only once message delivery is not guaranteed and it is possible that some messages will be lost or delivered twice.

11.1.33. Paging

The **`paging`** example shows how HornetQ can support huge queues even when the server is running in limited RAM. It does this by transparently *paging* messages to disk, and *depaging* them when they are required.

11.1.34. Pre-Acknowledge

Standard JMS supports three acknowledgment modes: **`AUTO_ACKNOWLEDGE`**, **`CLIENT_ACKNOWLEDGE`**, and **`DUPS_OK_ACKNOWLEDGE`**. For a full description on these modes please consult the JMS specification, or any JMS tutorial.

All of these standard modes involve sending acknowledgments from the client to the server. However in some cases, you really don't mind losing messages in event of failure, so it would make sense to acknowledge the message on the server before delivering it to the client. This example demonstrates how HornetQ allows this with an extra acknowledgment mode.

11.1.35. Message Producer Rate Limiting

The **`producer-rate-limit`** example demonstrates how, with HornetQ, you can specify a maximum send rate at which a JMS message producer will send messages.

11.1.36. Queue

A simple example demonstrating a JMS queue.

11.1.37. Message Redistribution

The **`queue-message-redistribution`** example demonstrates message redistribution between queues with the same name deployed in different nodes of a cluster.

11.1.38. Queue Requester

A simple example demonstrating a JMS queue requester.

11.1.39. Queue with Message Selector

The **`queue-selector`** example shows you how to selectively consume messages using message selectors with queue consumers.

11.1.40. Reattach Node example

The **`Reattach Node`** example shows how a client can try to reconnect to the same server instead of failing the connection immediately and notifying any user ExceptionListener objects. HornetQ can be configured to automatically retry the connection, and reattach to the server when it becomes available again across the network.

11.1.41. Request-Reply example

A simple example showing the JMS request-response pattern.

11.1.42. Scheduled Message

The **`scheduled-message`** example shows you how to send a scheduled message to a JMS Queue with HornetQ. Scheduled messages will not get delivered until a specified time in the future.

11.1.43. Security

The **`security`** example shows you how to configure and use role based queue security with HornetQ.

11.1.44. Send Acknowledgments

The **`send-acknowledgements`** example shows you how to use HornetQ's advanced *asynchronous send acknowledgments* feature to obtain acknowledgment from the server that sends have been received and processed in a separate stream to the sent messages.

11.1.45. SSL Transport

The **`ssl-enabled`** shows you how to configure SSL with HornetQ to send and receive message.

11.1.46. Static Message Selector

The **`static-selector`** example shows you how to configure a HornetQ core queue with static message selectors (filters).

11.1.47. Static Message Selector Using JMS

The **`static-selector-jms`** example shows you how to configure a HornetQ queue with static message selectors (filters) using JMS.

11.1.48. Stomp

The **`Stomp`** example shows you how to configure a HornetQ server to send and receive Stomp messages.

11.1.49. Stomp Over Web Sockets

The **`stomp-websockets`** example shows you how to configure a HornetQ server to send and receive Stomp messages directly from Web browsers (provided they support Web Sockets).

11.1.50. Symmetric Cluster

The **`symmetric-cluster`** example demonstrates a symmetric cluster set up with HornetQ.

HornetQ has extremely flexible clustering which allows you to set up servers in many different topologies. The most common topology that you'll perhaps be familiar with if you are used to application server clustering is a symmetric cluster.

With a symmetric cluster, the cluster is homogeneous, that is, each node is configured the same as every other node, and every node is connected to every other node in the cluster.

11.1.51. Temporary Queue

A simple example demonstrating how to use a JMS temporary queue.

11.1.52. Topic

A simple example demonstrating a JMS topic.

11.1.53. Topic Hierarchy

HornetQ supports topic hierarchies. With a topic hierarchy you can register a subscriber with a wild-card and that subscriber will receive any messages sent to an address that matches the wild card.

11.1.54. Topic Selector 1

The **`topic-selector-example1`** example shows you how to send message to a JMS topic, and subscribe them using selectors with HornetQ.

11.1.55. Topic Selector 2

The **`topic-selector-example2`** example shows you how to selectively consume messages using message selectors with topic consumers.

11.1.56. Transaction Failover With Data Replication

The **`transaction-failover`** example demonstrates two servers coupled as a live-backup pair for high availability (HA), and a client using a transacted JMS session failing over from live to backup when the live server is crashed.

HornetQ implements failover of client connections between live and backup servers. This is implemented by the replication of data between live and backup nodes. When replication is configured and a live node crashes, the client connections can carry and continue to send and consume messages. When transacted sessions are used, once only message delivery is guaranteed.

11.1.57. Transactional Session

The **`transactional`** example shows you how to use a transactional session with HornetQ.

11.1.58. XA Heuristic

The **`xa-heuristic`** example shows you how to make an XA heuristic decision through HornetQ Management Interface. A heuristic decision is a unilateral decision to commit or rollback an XA transaction branch after it has been prepared.

11.1.59. XA Receive

The **`xa-receive`** example shows you how message receiving behaves in an XA transaction in HornetQ.

11.1.60. XA Send

The **`xa-send`** example shows you how message sending behaves in an XA transaction in HornetQ.

11.1.61. XA with Transaction Manager

The **`xa-with-jta`** example shows you how to use JTA interfaces to control transactions with HornetQ.

11.2. Core API Examples

To run a core example, **`cd`** into the appropriate example directory and type **`ant`**

11.3. Java EE Examples

Most of the Java EE examples can be run the following way:

`cd` into the appropriate example directory and type **`./build.sh deploy`**. This will create a new server profile and start the server. When the server is started from a different window, type **`./build.sh run`** to run the example. Some of the examples required further steps; please refer to the examples documentation for more information.

11.3.1. EJB/JMS Transaction

An example that shows using an EJB and JMS together within a transaction.

11.3.2. HAJNDI (High Availability)

A simple example demonstrating using JNDI within a cluster.

11.3.3. Resource Adapter Configuration

This example demonstrates how to configure several properties on the HornetQ JCA resource adaptor.

11.3.4. Resource Adapter Remote Server Configuration

This example demonstrates how to configure the HornetQ resource adapter to talk to a remote HornetQ server

11.3.5. JMS Bridge

An example demonstrating the use of the HornetQ JMS bridge.

11.3.6. MDB (Message Driven Bean)

A simple example of a message driven bean.

11.3.7. Servlet Transport

An example of how to use the HornetQ servlet transport.

11.3.8. Servlet SSL Transport

An example of how to use the HornetQ servlet transport over SSL.

11.3.9. XA Recovery

An example of how XA recovery works within the JBoss Enterprise Application Platform using HornetQ.

Routing Messages With Wild Cards

HornetQ allows the routing of messages via wildcard addresses.

If a queue is created with an address of **`queue.news.#`**, for example, then it will receive any messages sent to addresses that match this. Take these, for example: **`queue.news.europe`** or **`queue.news.usa`** or **`queue.news.usa.sport`**. If you create a consumer on this queue, this allows a consumer to consume messages which are sent to a *hierarchy* of addresses.

Note

In JMS terminology this allows "topic hierarchies" to be created.

To enable this functionality set the property **`wild-card-routing-enabled`** in the **`hornetq-configuration.xml`** file to **`true`**. This is **`true`** by default.

For more information on the wild card syntax take a look at *Chapter 13, Understanding the HornetQ Wildcard Syntax* chapter, also see *Section 11.1.53, “Topic Hierarchy”*.

Understanding the HornetQ Wildcard Syntax

HornetQ uses a specific syntax for representing wildcards in security settings, address settings, and when creating consumers.

A HornetQ wildcard expression contains words delimited by the character '**.**' (full stop).

The special characters '**#**' and '*****' also have special meaning and can take the place of a word.

The character '**#**' means "match any sequence of zero or more words".

The character '*****' means "match a single word".

So the wildcard 'news.europe.#' would match 'news.europe', 'news.europe.sport', 'news.europe.politics', and 'news.europe.politics.regional' but would not match 'news.usa', 'news.usa.sport' or 'entertainment'.

The wildcard 'news.*' would match 'news.europe', but not 'news.europe.sport'.

The wildcard 'news.*.sport' would match 'news.europe.sport' and also 'news.usa.sport', but not 'news.europe.politics'.

Filter Expressions

HornetQ provides a powerful filter language based on a subset of the SQL 92 expression syntax.

It is the same as the syntax used for JMS selectors, but the predefined identifiers are different.

Filter expressions are used in several places in HornetQ.

- Predefined Queues. When pre-defining a queue, either in **`hornetq-configuration.xml`** or **`hornetq-jms.xml`** a filter expression can be defined for a queue. Only messages that match the filter expression will enter the queue.
- Core bridges can be defined with an optional filter expression, only matching messages will be bridged (see *Chapter 36, Core Bridges*).
- Diverts can be defined with an optional filter expression. Only matching messages will be diverted (see *Chapter 35, Diverting and Splitting Message Flows*).
- Filters are also used programmatically when creating consumers, queues and in several places as described in *Chapter 30, Management*.

There are some differences between JMS selector expressions and HornetQ core filter expressions. Whereas JMS selector expressions operate on a JMS message, HornetQ core filter expressions operate on a core message.

The following identifiers can be used in a core filter expression to refer to attributes of the core message in an expression:

HQPriority
: To refer to the priority of a message. Message priorities are integers with valid values from **`0 - 9`**. **`0`** is the lowest priority and **`9`** is the highest. For example, **`HQPriority = 3 and department = 'payroll'`**. This refers to a message with a priority of three and a department of 'payroll'.

HQExpiration
: To refer to the expiration time of a message. The value is a long integer.

HQDurable
: To refer to whether a message is durable or not. The value is a string with valid values: **`DURABLE`** or **`NON_DURABLE`**.

HQTimestamp
: The timestamp of when the message was created. The value is a long integer.

HQSize
: The size of a message in bytes. The value is an integer.

Any other identifiers used in core filter expressions will be assumed to be properties of the message.

Persistence

This chapter covers persistence and its configuration in HornetQ.

HornetQ handles its own persistence. It ships with a high-performance journal, which is optimized for messaging-specific use cases.

The HornetQ journal is *append only* with a configurable file size, which improves performance by enabling single write operations. It consists of a set of files on disk, which are initially pre-created to a fixed size and filled with padding. As server operations (add message, delete message, update message, etc.) are performed, records of the operations are appended to the journal until the journal file is full, at which point the next journal file is used.

A sophisticated garbage collection algorithm determines whether a journal files can be reclaimed and re-used when all of their data has been deleted. A compaction algorithm removes dead space from journal files and compresses the data.

The journal also fully supports both local and XA transactions.

The majority of the journal is written in Java, but interaction with the file system has been abstracted to allow different pluggable implementations. The two implementations shipped with HornetQ are:

Java Non-blocking IO (NIO)
: Uses standard Java NIO to interface with the file system. This provides extremely good performance and runs on any platform with a Java 6 or later runtime.

Linux Asynchronous IO (AIO)
: Uses a native code wrapper to talk to the Linux asynchronous IO library (AIO). With AIO, HornetQ receives a message when data has been persisted. This removes the need for explicit syncs. AIO will typically provide better performance than Java NIO, but requires Linux kernel 2.6 or later and **libaio**.

  AIO also requires **`ext2`**, **`ext3`** **`ext4`**, **`jfs`** or **`xfs`** type file systems. On NFS, it appears to work correctly, but falls back to slower, synchronous behavior.

Installing AIO

On Red Hat Enterprise Linux, install **libaio** with the following command:

```
yum install libaio
```

The standard HornetQ core server uses two instances of the journal:

bindings journal
: Stores bindings-related data, including the set of queues deployed on the server and their attributes. It also stores data such as ID sequence counters. The bindings journal is always a NIO journal, as it typically has low throughput in comparison to the message journal.

  The files on this journal are prefixed as **`hornetq-bindings`**. Each file has a **`bindings`** extension. File size is **`1048576`**, and it is located at the bindings folder.

JMS journal
: Stores all JMS-related data, for example, any JMS queues, topics or connection factories and any JNDI bindings for these resources. Any JMS resources created with the management API are

persisted to this journal. Any resources configures with configuration files are not. This journal is created only if JMS is in use.

message journal
: Stores all message-related data, including messages themselves and duplicate-id caches. By default, HornetQ uses AIO for this journal. If AIO is not available, it will automatically fall back to NIO.

Large messages are persisted outside the message journal. For more information see *Chapter 23, Large Messages*.

In low memory situations, configure HornetQ to page messages to disk. See *Chapter 24, Paging* for more information.

If persistence is not required, HornetQ can be configured not to persist any data, as discussed in *Section 15.4, "Configuring HornetQ for Zero Persistence"*.

15.1. Configuring the bindings journal

The bindings journal is configured using the following attributes in **`hornetq-configuration.xml`**.

`bindings-directory`
: The location of the bindings journal. The default value is **`data/bindings`**.

`create-bindings-dir`
: If **`true`**, and the bindings directory does not exist, the bindings directory is created automatically at the location specified in `bindings-directory`. The default value is **`true`**.

15.2. Configuring the JMS journal

The JMS journal shares its configuration with the bindings journal.

15.3. Configuring the message journal

The message journal is configured using the following attributes in **`hornetq-configuration.xml`**.

`journal-directory`
: The location of the message journal. The default value is **`data/journal`**. For best performance, this journal should be located on its own physical volume to minimize disk head movement. If this journal is stored on a storage area network, each journal instance on the network should have its own logical unit.

`create-journal-dir`
: If **`true`**, the journal directory is created at the location specified in `journal-directory`. The default value is **`true`**.

`journal-type`
: Valid values are **`NIO`** or **`ASYNCIO`**. If **`NIO`**, the Java NIO journal is used. If **`ASYNCIO`**, Linux asynchronous IO is used. If **`ASYNCIO`** is set on a non-Linux or non-libaio system, HornetQ detects this and falls back to **`NIO`**.

`journal-sync-transactional`
: If **`true`**, HornetQ ensures all transaction data is flushed to disk on transaction boundaries (commit, prepare, and rollback). The default is **`true`**.

`journal-sync-non-transactional`
If **`true`**, HornetQ ensures non-transactional message data (sends and acknowledgments) are flushed to disk. The default is **`true`**.

`journal-file-size`
The size of each journal file in bytes. The default value is **`10485760`** bytes (10 megabytes).

`journal-min-files`
The minimum number of files the journal maintains. When HornetQ starts and there is no initial data, HornetQ pre-creates this number of files. Creating and padding journal files is an expensive operation, so to be avoided at run-time as files are filled. Pre-creating files means that as one is filled the journal can immediately resume with the next file without pausing to create it.

`journal-max-io`
The maximum number of write requests to hold in the IO queue. Write requests are queued here before being submitted to the system for execution. If the queue fills, writes are blocked until space becomes available in the queue. For NIO, this must be **`1`**. For AIO, this should be **`500`**. A different default value is maintained depending on whether NIO or AIO is used (**`1`** for NIO, **`500`** for AIO). The total max AIO must not be higher than what is configured at the operating system level (**`/proc/sys/fs/aio-max-nr`**), generally at 65536.

`journal-buffer-timeout`
HornetQ maintains a buffer of flush requests, and flushes the entire buffer either when it is full or when this timeout expires - whichever is soonest. This is used for both NIO and AIO and allows improved scaling when many concurrent writes and flushes are required.

`journal-buffer-size`
The size of the timed buffer on AIO. The default value is **`490`** kilobytes.

`journal-compact-min-files`
The minimum number of files before the journal will be compacted. The default value is **`10`**.

`journal-compact-percentage`
When less than this percentage of a journal is considered live data compacting will occur. The default value is **`30`**. `journal-compact-min-files` must also be fulfilled before compacting.

Disabling disk write cache

Most disks contain hardware write caches, which increase the apparent performance of a disk because writes are cached and lazily written to disk later.

Many systems ship with disk write cache enabled by default, so even after syncing from the operating system there is no guarantee that the data has been written to disk. If a failure occurs, critical data can still be lost.

Some systems have non-volatile or battery-backed write caches. These will not necessarily lose data in the event of failure, but testing is essential.

If your disk does not have these backups in place, and is not part of a redundant array (for example, RAID), ensure that disk write cache is **`disabled`**. This can have negative effects on performance, but ensures data integrity.

On Linux, inspect or change your disk's write cache settings with the **hdparm** tool for IDE disks, or **sdparm** or **sginfo** tools for SDSI or SATA disks.

On Windows, check or change settings by right-clicking on the disk and selecting **Properties**.

15.4. Configuring HornetQ for Zero Persistence

To configure HornetQ to perform zero persistence, set the `persistence-enabled` parameter in **`hornetq-configuration.xml`** to **`false`**.

Zero Persistence

Once this parameter is set to **`false`**, no persistence will occur. No bindings data, message data, large message data, duplicate ID caches or paging data will be persisted.

15.5. Import/Export the Journal Data

You may want to inspect the existent records on each one of the journals used by HornetQ, and you can use the export/import tool for that purpose. The export/import are classes located at the hornetq-core.jar, you can export the journal as a text file by using this command:

```
java -cp hornetq-core.jar org.hornetq.core.journal.impl.ExportJournal
<JournalDirectory> <JournalPrefix> <FileExtension> <FileSize> <FileOutput>
```

To import the file as binary data on the journal (Notice you also require netty.jar):

```
java -cp hornetq-core.jar:netty.jar
org.hornetq.core.journal.impl.ImportJournal <JournalDirectory>
<JournalPrefix> <FileExtension> <FileSize> <FileInput>
```

- JournalDirectory: Use the configured folder for your selected folder. Example: ./hornetq/data/journal
- JournalPrefix: Use the prefix for your selected journal, as discussed
- FileExtension: Use the extension for your selected journal, as discussed
- FileSize: Use the size for your selected journal, as discussed
- FileOutput: text file that will contain the exported data

Configuring the Transport

HornetQ has a fully pluggable and highly flexible transport layer. The transport layer defines its own Service Provider Interface (SPI) to simplify plugging in a new transport provider.

This chapter covers the concepts required to use and configure HornetQ transports.

16.1. Understanding Acceptors

Acceptors are defined in the **`hornetq-configuration.xml`** configuration file:

```
<acceptors>
   <acceptor name="netty">
      <factory-class> org.hornetq.core.remoting.impl.netty.NettyAcceptorFactory</factory-
class>
      <param key="port" value="5446"/>
   </acceptor>
</acceptors>
```

Acceptors are always defined inside an **`acceptors`** element. Multiple acceptors can be defined in one **`acceptors`** element. There is no upper limit to the number of acceptors per server.

Each acceptor defines a way in which connections can be made to the HornetQ server.

The above example defines an **`acceptor`** that uses Netty to listen for connections on port **`5446`**.

The **`acceptor`** element contains a sub-element **`factory-class`** which defines the factory used to create acceptor instances. In this case Netty is used to listen for connections, so the Netty implementation of **`AcceptorFactory`** is being used. The **`factory-class`** element determines which pluggable transport listens.

The **`acceptor`** element can also be configured with zero or more **`param`** sub-elements. Each **`param`** element defines a key-value pair. These key-value pairs are used to configure the specific transport, the set of valid key-value pairs depends on the specific transport be used and are passed straight through to the underlying transport.

Examples of key-value pairs for a particular transport would be, say, to configure the IP address to bind to, or the port to listen at.

16.2. Understanding Connectors

Where acceptors are used on the server to define how we accept connections, connectors are used by a client to define how it connects to a server. Connectors are also defined in the **`hornetq-configuration.xml`** file:

```
<connectors>
   <connector name="netty">
      <factory-class>org.hornetq.core.remoting.impl.netty.NettyConnectorFactory</factory-
class>
      <param key="port" value="5446"/>
   </connector>
</connectors>
```

Connectors can be defined inside a **`connectors`** element. Multiple connectors can be defined in the same **`connectors`** element. There is no upper limit to the number of connectors per server.

Although they are used by the client, they are defined on the server for a number of reasons:

- Sometimes the server acts as a client itself when it connects to another server, for example when one server is bridged to another, or when a server takes part in a cluster. In this cases the server needs to know how to connect to other servers. This is defined by *connectors*.
- If JMS and the server-side JMS service are used to instantiate JMS ConnectionFactory instances and bind them in JNDI, then the JMS service needs to know which server the **`HornetQConnectionFactory`** will create connections to at the connection factory's creation. This is defined by the **`connector-ref`** element in the **`hornetq-jms.xml`** file on the server side. Let's take a look at a snippet from a **`hornetq-jms.xml`** file that shows a JMS connection factory that references our netty connector defined in our **`hornetq-configuration.xml`** file:

```
<connection-factory name="ConnectionFactory">
    <connectors>
       <connector-ref connector-name="netty"/>
    </connectors>
    <entries>
        <entry name="ConnectionFactory"/>
        <entry name="XAConnectionFactory"/>
    </entries>
</connection-factory>
```

16.3. Configuring the transport directly from the client side

This section shows how to configure a core **`ClientSessionFactory`** to connect with a server.

Connectors are used indirectly when configuring a core **`ClientSessionFactory`** to talk to a server. In this case, it is unnecessary to define a connector in the server-side configuration. Instead, create the parameters and configure the connector factory to be used by **`ClientSessionFactory`**.

The following **`ClientSessionFactory`** connects directly to the acceptor defined previously in this chapter. It uses the standard Netty TCP transport, and will attempt to connect on port 5446 to localhost (the default).

```
Map<String, Object> connectionParams =
  new HashMap<String, Object>();

  connectionParams.put(
    org.hornetq.core.remoting.impl.netty.TransportConstants.PORT_PROP_NAME,
    5446
  );

TransportConfiguration transportConfiguration =
  new TransportConfiguration(
    "org.hornetq.core.remoting.impl.netty.NettyConnectorFactory",
    connectionParams
  );

ClientSessionFactory sessionFactory =
  HornetQClient.createClientSessionFactory(transportConfiguration);

ClientSession session = sessionFactory.createSession(...);
```

For JMS, you can configure the JMS connection directly on the client side without defining a connector on the server side or a connection factory in **`hornetq-jms.xml`**:

```
Map<String, Object> connectionParams =
```

```
  new HashMap<String, Object>();

connectionParams.put(
  org.hornetq.core.remoting.impl.netty.TransportConstants.PORT_PROP_NAME,
  5446
);

TransportConfiguration transportConfiguration =
  new TransportConfiguration(
    "org.hornetq.core.remoting.impl.netty.NettyConnectorFactory",
    connectionParams
  );

ConnectionFactory connectionFactory =
  HornetQJMSClient.createConnectionFactory(transportConfiguration);

Connection jmsConnection = connectionFactory.createConnection();
```

16.4. Configuring the Netty transport

Out of the box, HornetQ uses Netty, a high-performance, low-level network library.

The Netty transport can be configured to use old (blocking) Java IO, NIO (non-blocking), TCP sockets, SSL, HTTP or HTTPS. A servlet transport is also provided.

16.4.1. Configuring Netty TCP

Netty TCP is a simple unencrypted TCP socket-based transport. Netty TCP can be configured to use blocking Java Asynchronous IO (AIO) or non-blocking Java NIO (NIO). NIO is recommended on the server-side for concurrent connection scalability; however, AIO can provide better latency if many concurrent connections are not required.

This transport is unencrypted

If you are running connections across an untrusted network, consider SSL or HTTPS instead.

With the Netty TCP transport all connections are initiated from the client side. This is useful in situations where firewall policies only allow connections to be initiated in one direction.

All valid Netty transport keys are defined in **`org.hornetq.core.remoting.impl.netty.TransportConstants`**. Most parameters can be used with acceptors and connectors. Some only work with acceptors. The following parameters can be used to configure Netty for simple TCP:

`use-nio`
: If **`true`**, NIO is used. If **`false`**, AIO is used. The default is **`false`** on both client and server sides. If your server must handle many concurrent connections, NIO is recommended, since it does not maintain one thread per connection and can scale to many more concurrent connections than AIO. If your server does not need to handle many concurrent connections, AIO may provide better performance.

`host`
: The host name or IP address to connect to (for connectors) or listen on (for acceptors). The default value is **`localhost`**.

Configuration Required

The default for this variable is **`localhost`**. This is not accessible from remote nodes and must be modified for the server to accept incoming connections.

Acceptors can be configured with multiple comma-delimited hosts or IP addresses. Multiple addresses are not valid for connectors. **`0.0.0.0`** specifies that all network interfaces of a host should be accepted.

`port`
: Specifies the port to connect to (for connectors) or listen on (for acceptors). The default value is **`5445`**.

`tcp-no-delay`
: If **`true`**, Nagle's algorithm is enabled. The default value is **`true`**.

`tcp-send-buffer-size`
: Defines the size of the TCP send buffer in bytes. The default value is **`32768`** (32 kilobytes). TCP buffer size should be tuned according to the bandwidth and latency of your network. Put simply, the buffer size in bytes should be equal to the bandwidth in bytes-per-second multiplied by the network round-trip-time (RTT) in seconds. RTT can be measured using the **ping** utility. For fast networks, you may wish to increase the buffer size from the default value.

`tcp-receive-buffer-size`
: Defines the size of the TCP receive buffer in bytes. The default value is **`32768`** (32 kilobytes).

`batch-delay`
: HornetQ can be configured to place write operations into batches for up to `batch-delay` milliseconds. This can increase overall throughput for very small messages, but does so at the expense of an increase in average latency for message transfer. The default value is **`0`** milliseconds.

`direct-deliver`
: When a message arrives on the server and is delivered to consumers, by default the delivery occurs on a different thread to that in which the message arrived. This gives the best overall throughput and scalability, especially on multi-core machines. However, it also introduces additional latency due to the context switch required. For the lowest latency (and possible reduction of throughput), set `direct-deliver` to **`true`** (the default). For highest throughput, set to **`false`**.

`nio-remoting-threads`
: When configured to use NIO, by default HornetQ uses three times the number of threads as cores (or hyper-threads) reported by `Runtime.getRuntime().availableProcessors()` to process incoming packets. `nio-remoting-threads` overrides this and defines the number of threads to use. The default is **`-1`**, which represents three times the value from `Runtime.getRuntime().availableProcessors()`.

16.4.2. Configuring Netty SSL

Netty SSL is similar to the Netty TCP transport but it provides additional security by encrypting TCP connections using the Secure Sockets Layer (SSL).

See the HornetQ examples for a working example of Netty SSL.

Netty SSL uses the same properties as Netty TCP in addition to the following properties:

`ssl-enabled`
Set to **`true`** to enable SSL.

`key-store-path`
Defines the path to the SSL key store on the client that holds the client certificates.

`key-store-password`
Defines the password for the client certificate key store on the client.

`trust-store-path`
Defines the path to the trusted client certificate store on the server.

`trust-store-password`
Defines the password to the trusted client certificate store on the server.

16.4.3. Configuring Netty HTTP

Netty HTTP tunnels packets over the HTTP protocol. It can be useful in scenarios where firewalls only allow HTTP traffic to pass.

See the HornetQ examples for a working example of Netty HTTP.

Netty HTTP uses the same properties as Netty TCP but adds the following additional properties:

`http-enabled`
Set to **`true`** to enable HTTP.

`http-client-idle-time`
The period of time (in milliseconds) a client can be idle before sending an empty HTTP request to keep the connection alive.

`http-client-idle-scan-period`
How often, in milliseconds, to scan for idle clients.

`http-response-time`
The period of time in milliseconds the server should wait before sending and empty HTTP response to keep the connection alive.

`http-server-scan-period`
How often, in milliseconds, to scan for clients requiring responses.

`http-requires-session-id`
When **`true`**, the client waits to receive a session ID after the first call. This is used when the HTTP connector is connecting to the servlet acceptor (not recommended).

16.4.4. Configuring Netty Servlet

Netty Servlet transport allows HornetQ traffic to be tunneled over HTTP to a servlet running in a servlet engine, which redirects it to an in-virtual machine HornetQ server.

This differs from the Netty HTTP transport in that traffic is routed through a servlet engine which may already be serving web applications. This allows HornetQ to be used where corporate policies allow only a single web server to listen on an HTTP port. See the HornetQ examples for a working example of the Netty Servlet transport.

Configuring a servlet engine for the Netty Servlet transport

1. Deploy the servlet. A web application using the servlet might have a **`web.xml`** file similar to the following:

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="http://java.sun.com/xml/ns/j2ee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee http://java.sun.com/xml/ns/j2ee/
web-app_2_4.xsd"
  version="2.4">
  <servlet>
    <servlet-name>HornetQServlet</servlet-name>
    <servlet-class>org.jboss.netty.channel.socket.http.HttpTunnelingServlet</servlet-
class>
    <init-param>
      <param-name>endpoint</param-name>
      <param-value>local:org.hornetq</param-value>
    </init-param>
    <load-on-startup>1</load-on-startup>
  </servlet>

  <servlet-mapping>
    <servlet-name>HornetQServlet</servlet-name>
    <url-pattern>/HornetQServlet</url-pattern>
  </servlet-mapping>
</web-app>
```

2. Add the **`netty-invm`** acceptor to the server-side configuration in **`hornetq-configuration.xml`**:

```
<acceptors>
  <acceptor name="netty-invm">
    <factory-class>
      org.hornetq.core.remoting.impl.netty.NettyAcceptorFactory
    </factory-class>
    <param key="use-invm" value="true"/>
    <param key="host" value="org.hornetq"/>
  </acceptor>
</acceptors>
```

3. Define a connector for the client in **`hornetq-configuration.xml`**:

```
<connectors>
  <connector name="netty-servlet">
    <factory-class>
      org.hornetq.core.remoting.impl.netty.NettyConnectorFactory
    </factory-class>
    <param key="host" value="localhost"/>
    <param key="port" value="8080"/>
    <param key="use-servlet" value="true"/>
    <param key="servlet-path" value="/messaging/HornetQServlet"/>
  </connector>
</connectors>
```

Init Parameters

`endpoint`

Defines the netty acceptor to which the servlet forwards its packets. Matches the name of the `host` parameter.

The servlet pattern configured in the **`web.xml`** is the path of the URL that is used. The connector parameter `servlet-path` on the connector configuration must match this using the application context of the web application if there is one.

The servlet transport can also be used over SSL by adding the following configuration to the connector:

```
<connector name="netty-servlet">
  <factory-class>
    org.hornetq.core.remoting.impl.netty.NettyConnectorFactory
  </factory-class>
  <param key="host" value="localhost"/>
  <param key="port" value="8443"/>
  <param key="use-servlet" value="true"/>
  <param key="servlet-path" value="/messaging/HornetQServlet"/>
  <param key="ssl-enabled" value="true"/>
  <param key="key-store-path" value="path to a keystore"/>
  <param key="key-store-password" value="keystore password"/>
</connector>
```

You will also have to configure the application server to use a KeyStore. Edit the SSL/TLS connector configuration in **`server/default/deploy/jbossweb.sar/server.xml`** like so:

```
<Connector protocol="HTTP/1.1" SSLEnabled="true"
  port="8443" address="${jboss.bind.address}"
  scheme="https" secure="true" clientAuth="false"
  keystoreFile="path to a keystore"
  keystorePass="keystore password" sslProtocol = "TLS" />
```

In both cases you will need to provide a keystore and password. See the Servlet SSL example shipped with HornetQ for more detail.

Detecting Dead Connections

This chapter discusses Connection Time-to-Live (TTL), and explains how HornetQ deals with crashed clients and clients that have exited without cleanly closing their resources.

17.1. Cleaning up Dead Connection Resources on the Server

Before a HornetQ client application exits, its resources should be closed using a **`finally`** block.

Example 17.1. Well Behaved Core Client Application

Below is an example of a well behaved core client application closing its session and session factory in a finally block:

```
ClientSessionFactory sf = null;
ClientSession session = null;

try
{
   sf = HornetQClient.createClientSessionFactory(...);
   session = sf.createSession(...);
   ... do some stuff with the session...
}
finally
{
   if (session != null)
   {
      session.close();
   }

   if (sf != null)
   {
      sf.close();
   }
}
```

Example 17.2. Well Behaved JMS Client Application

This is an example of a well behaved JMS client application:

```
Connection jmsConnection = null;

try
{
   ConnectionFactory jmsConnectionFactory = HornetQJMSClient.createConnectionFactory(...);

   jmsConnection = jmsConnectionFactory.createConnection();

   ... do some stuff with the connection...
}
finally
{
   if (connection != null)
   {
      connection.close();
   }
}
```

If a client crashes, server side resources like sessions can be left hanging on the server. This can cause a resource leak over time and lead to the server running out of memory or other resources.

The requirement for cleaning up dead client resources is balanced with the HornetQ's support of client reconnection. The network between the client and the server can fail and then reboot, allowing the client to reconnect. If the resources have been cleaned, the reboot will fail.

To ensure that resources are cleaned upon client crash, while allowing for reboot time, the `connection TTL` is configurable. Each **`ClientSessionFactory`** has a defined `connection TTL`. The TTL determines how long the server will keep a connection alive in the absence of any data arriving from the client.

The client automatically sends ping packets to prevent the server from closing it down. If the server does not receive any packets on a connection for the connection TTL time, then it will close all the sessions on the server that relate to that connection.

For JMS, the connection TTL is defined by the `ConnectionTTL` attribute on a **`HornetQConnectionFactory`** instance. If JMS connection factory instances are being deployed directly into JNDI on the server side, the connection TTL can be specified in the xml config, using the parameter `connection-ttl`.

The default value for connection TTL is 60000ms, (1 minute). A value of **`-1`** for `ConnectionTTL` means the server will never time out the connection on the server side.

To prevent clients specifying their own connection TTL, a global value can be set on the server side that overrides all other values. To do this, specify the `connection-ttl-override` attribute in the server side configuration. The default value for `connection-ttl-override` is **`-1`**, which allows clients to set their own values for connection TTL.

17.1.1. Closing core sessions or JMS connections that have failed to close

It is important that all core client sessions and JMS connections are always closed explicitly in a **`finally`** block when they are finished.

If a session or connection is not closed in a **`finally`** block, HornetQ will detect this at garbage collection time, and log a warning similar to the following in the logs (If you are using JMS the warning will involve a JMS connection not a client session):

```
[Finalizer] 20:14:43,244 WARNING [org.hornetq.core.client.impl.DelegatingSession]
  I'm closing a ClientSession you left open. Please make sure you close all ClientSessions
  explicitly before letting them go out of scope!
[Finalizer] 20:14:43,244 WARNING [org.hornetq.core.client.impl.DelegatingSession]
The session you didn't close was created here:
java.lang.Exception
  at org.hornetq.core.client.impl.DelegatingSession.<init>(DelegatingSession.java:83)
  at org.acme.yourproject.YourClass (YourClass.java:666)
```

HornetQ will then close the connection / client session.

The log will also print the line of user code where the JMS connection/client session that did not close was created. This enables the error to be pinpointed and corrected appropriately.

17.2. Detecting failure from the client side.

The client pings the server to prevent the server cleaning dead resources. It also pings the server to detect if the server or network has failed.

As long as the client is receiving data from the server it will consider the connection to still be alive.

If the client does not receive any packets for `client-failure-check-period` milliseconds then it will consider the connection failed and will either initiate failover, or call any `FailureListener` instances (or `ExceptionListener` instances if you are using JMS) depending on how it has been configured.

For JMS, the check period is defined by the `ClientFailureCheckPeriod` attribute on a **`HornetQConnectionFactory`** instance. If JMS connection factory instances are being deployed directly into JNDI on the server side, it can be specified in the **`hornetq-jms.xml`** configuration file, using the parameter `client-failure-check-period`.

The default value for the client failure check period is **`30000`**ms (30 seconds). A value of **`-1`** means the client will never fail the connection on the client side if no data is received from the server. Typically this is much lower than the connection TTL value to allow clients to reconnect in case of transitory failure.

17.3. Configuring Asynchronous Connection Execution

By default, packets received on the server side are executed on the remoting thread.

A thread from a thread pool can be used instead to handle some packets so that the remoting thread is not tied up for too long.

Note

Processing operations asynchronously on another thread adds a little more latency.

Most short running operations are handled on the remoting thread for performance reasons. Ensure the parameter `async-connection-execution-enabled` in **`hornetq-configuration.xml`** is set to true to enable asynchronous connection execution.

Resource Manager Configuration

HornetQ has its own Resource Manager for handling the lifespan of JTA transactions. When a transaction is started the Resource Manager is notified and keeps a record of the transaction and its current state. Sometimes, a transaction will be started and then forgotten. If this happens, the transaction will sit indefinitely. If configured, HornetQ can scan for old transactions and rollback any expired transactions. The default lifespan of a transaction is five minutes (or 3000000 milliseconds). That is, any transactions older than five minutes are removed.

This timeout lifespan can be changed by editing the **`transaction-timeout`** property in **`hornetq-configuration.xml`** (value must be in milliseconds). The property **`transaction-timeout-scan-period`** configures how often, in milliseconds, to scan for old transactions.

Please note that HornetQ will not unilaterally rollback any XA transactions in a prepared state - this must be heuristically rolled back via the management API if you are sure they will never be resolved by the transaction manager.

Flow Control

Flow control is used to limit the flow of data between a client and server, or a server and another server. It does this in order to prevent the client or server being overwhelmed with data.

19.1. Consumer Flow Control

This controls the flow of data between the server and the client as the client consumes messages. For performance reasons clients normally buffer messages before delivering to the consumer via the **`receive()`** method or asynchronously via a message listener. Messages can build up if the consumer cannot process messages as fast as they are being delivered and installed on the internal buffer. This can potentially lead to a lack of memory on the client if they cannot be processed in time.

19.1.1. Window-Based Flow Control

HornetQ consumers improve performance by buffering a certain number of messages in a client-side buffer before passing them to the client to consume.

To prevent a network round trip for every message, HornetQ pre-fetches messages into a buffer on each consumer. The total maximum size of messages (in bytes) that will be buffered on each consumer is determined by the **`consumer-window-size`** parameter.

By default, the **`consumer-window-size`** is set to 1 MiB (1024 * 1024 bytes).

The value can be:

- **`-1`** for an *unbound* buffer
- **`0`** to not buffer any messages. See *Section 11.1.31, "No Consumer Buffering"* for a working example of a consumer with no buffering.
- **`>0`** for a buffer with the given maximum size in bytes.

Setting the consumer window size can considerably improve performance depending on the messaging use case. As an example, consider the two extremes:

Fast consumers
: Fast consumers can process messages as fast as they consume them.

  To allow fast consumers, set the **`consumer-window-size`** to -1. This will allow *unbound* message buffering on the client side.

  Use this setting with caution: it can overflow the client memory if the consumer is not able to process messages as fast as it receives them.

Slow consumers
: Slow consumers take significant time to process each message and it is desirable to prevent buffering messages on the client side so that they can be delivered to another consumer instead.

  Consider a situation where a queue has two consumers; one of which is very slow. Messages are delivered in a circular fashion to both consumers; the fast consumer processes all of its messages very quickly until its buffer is empty. At this point there are still messages waiting to be processed in the buffer of the slow consumer which prevents them being processed by the fast consumer. The fast consumer is therefore sitting idle when it could be processing the other messages.

To allow slow consumers, set the **`consumer-window-size`** to 0 (for no buffer at all). This will prevent the slow consumer from buffering any messages on the client side. Messages will remain on the server side ready to be consumed by other consumers.

Setting this to 0 can give deterministic distribution between multiple consumers on a queue.

Most of the consumers cannot be clearly identified as fast or slow consumers but are in-between. In that case, setting the value of **`consumer-window-size`** to optimize performance depends on the messaging use case and requires benchmarks to find the optimal value, but a value of 1MiB is fine in most cases.

19.1.1.1. Using Core API

If HornetQ Core API is used, the consumer window size is specified by **`ClientSessionFactory.setConsumerWindowSize()`** method and some of the **`ClientSession.createConsumer()`** methods.

19.1.1.2. Using JMS

If JNDI is used to look up the connection factory, the consumer window size is configured in **`hornetq-jms.xml`**:

```
<connection-factory name="ConnectionFactory">
   <connectors>
      <connector-ref connector-name="netty-connector"/>
   </connectors>
   <entries>
      <entry name="ConnectionFactory"/>
   </entries>

   <!-- Set the consumer window size to 0 to have *no* buffer on the client side -->
   <consumer-window-size>0</consumer-window-size>
</connection-factory>
```

If the connection factory is directly instantiated, the consumer window size is specified by **`HornetQConnectionFactory.setConsumerWindowSize()`** method.

Please see *Section 11.1.31, "No Consumer Buffering"* for an example which shows how to configure HornetQ to prevent consumer buffering when dealing with slow consumers.

19.1.2. Rate limited flow control

It is also possible to control the *rate* at which a consumer can consume messages. This can be used to make sure that a consumer never consumes messages at a rate faster than the rate specified.

The rate must be a positive integer to enable this functionality and is the maximum desired message consumption rate specified in units of messages per second. Setting this to **`-1`** disables rate limited flow control. The default value is **`-1`**.

Please see *Section 11.1.10, "Message Consumer Rate Limiting"* for a working example of limiting consumer rate.

19.1.2.1. Using Core API

If the HornetQ core API is being used, the rate can be set via the **`ClientSessionFactory.setConsumerMaxRate(int consumerMaxRate)`** method or alternatively via some of the **`ClientSession.createConsumer()`** methods.

19.1.2.2. Using JMS

If JNDI is used to look up the connection factory, the max rate can be configured in **`hornetq-jms.xml`**:

```
<connection-factory name="ConnectionFactory">
   <connectors>
      <connector-ref connector-name="netty-connector"/>
   </connectors>
   <entries>
      <entry name="ConnectionFactory"/>
   </entries>
<!-- We limit consumers created on this connection factory to consume messages at a maximum
 rate of 10 messages per sec -->
   <consumer-max-rate>10</consumer-max-rate>
</connection-factory>
```

If the connection factory is directly instantiated, the max rate size can be set via the **`HornetQConnectionFactory.setConsumerMaxRate(int consumerMaxRate)`** method.

Note

Rate limited flow control can be used in conjunction with window based flow control. Rate limited flow control only effects how many messages a client can consume in a second and not how many messages are in its buffer. So if you had a slow rate limit and a high window based limit the clients internal buffer would soon fill up with messages.

Please see *Section 11.1.10, "Message Consumer Rate Limiting"* for an example which shows how to configure HornetQ to prevent consumer buffering when dealing with slow consumers.

19.2. Producer flow control

HornetQ also can limit the amount of data sent from a client to a server to prevent the server being overwhelmed.

19.2.1. Window based flow control

In a similar way to consumer window based flow control, HornetQ producers, by default, can only send messages to an address as long as they have sufficient credits to do so. The amount of credits required to send a message is given by the size of the message.

As producers run low on credits they request more from the server. When the server sends them more credits they can send more messages.

The amount of credits a producer requests in one go is known as the *window size*.

The window size therefore determines the amount of bytes that can be in-flight at any one time before more need to be requested; this prevents the remoting connection from getting overloaded.

19.2.1.1. Using Core API

If the HornetQ core API is being used, window size can be set via the **`ClientSessionFactory.setProducerWindowSize(int producerWindowSize)`** method.

19.2.1.2. Using JMS

If JNDI is used to look up the connection factory, the producer window size can be configured in **`hornetq-jms.xml`**:

```
<connection-factory name="ConnectionFactory">
   <connectors>
     <connector-ref connector-name="netty-connector"/>
   </connectors>
   <entries>
      <entry name="ConnectionFactory"/>
   </entries>
   <producer-window-size>10</producer-window-size>
</connection-factory>
```

If the connection factory is directly instantiated, the producer window size can be set via the **`HornetQConnectionFactory.setProducerWindowSize(int producerWindowSize)`** method.

19.2.1.3. Blocking producer window based flow control

Normally the server will always give the same number of credits as has been requested. However, it is also possible to set a maximum size on any address. This then blocks how many credits can be sent to the address so that its memory cannot be exceeded.

For example, if there is a JMS queue called "myqueue", the maximum memory size could be set to 10 megabytes, and the the server will control the number of credits sent to any producers which are sending any messages to myqueue. This means that the total messages in the queue never exceeds 10 megabytes.

When the address gets full, producers will block on the client side until more space frees up on the address; that is, until messages are consumed from the queue thus freeing up space for more messages to be sent. This is called blocking producer flow control.

To configure an address with a maximum size and tell the server that you want to block producers for this address if it becomes full, you need to define an AddressSettings (*Section 25.3, “Configuring Queues Through Address Settings”*) block for the address and specify **`max-size-bytes`** and **`address-full-policy`**

The address block applies to all queues registered to that address. That is, the total memory for all queues bound to that address will not exceed **`max-size-bytes`**. In the case of JMS topics this means the *total* memory of all subscriptions in the topic will not exceed max-size-bytes.

Here's an example:

```
<address-settings>
   <address-setting match="jms.queue.exampleQueue">
      <max-size-bytes>100000</max-size-bytes>
      <address-full-policy>BLOCK</address-full-policy>
   </address-setting>
</address-settings>
```

The above example would set the max size of the JMS queue "exampleQueue" to be 100000 bytes and would block any producers sending to that address to prevent that max size being exceeded.

Note the policy must be set to **`BLOCK`** to enable blocking producer flow control.

Note

Note that in the default configuration all addresses are set to block producers after 10 megabytes of message data is in the address. This means you cannot send more than 10MiB of message data to an address without it being consumed before the producers will be blocked. If this behavior is not required, increase the **`max-size-bytes`** parameter or change the address full message policy.

19.2.2. Rate limited flow control

The rate a producer can emit messages can be limited, in units of messages per second. This means that the producer will never produce messages at a rate higher than what has been set.

The rate must be a positive integer to enable this functionality and is the maximum desired message consumption rate specified in units of messages per second. Setting this to **`-1`** disables rate limited flow control. The default value is **`-1`**.

Please see the *Section 11.1.35, "Message Producer Rate Limiting"* for a working example of limiting producer rate.

19.2.2.1. Using Core API

If the HornetQ core API is being used, the rate can be set via the **`ClientSessionFactory.setProducerMaxRate(int consumerMaxRate)`** method or alternatively via some of the **`ClientSession.createProducer()`** methods.

19.2.2.2. Using JMS

If JNDI is used to look up the ConnectionFactory, the max rate can be configured in **`hornetq-jms.xml`**:

```
<connection-factory name="ConnectionFactory">
   <connectors>
      <connector-ref connector-name="netty-connector"/>
   </connectors>
   <entries>
      <entry name="ConnectionFactory"/>
   </entries>
<!-- We limit producers created on this connection factory to produce messages at a maximum
 rate of 10 messages per sec -->
   <producer-max-rate>10</producer-max-rate>
</connection-factory>
```

If the connection factory is directly instantiated, the max rate size can be set via the **`HornetQConnectionFactory.setProducerMaxRate(int consumerMaxRate)`** method.

Guarantees of sends and commits

20.1. Guarantees of Transaction Completion

When committing or rolling back a transaction with HornetQ, the request to commit or rollback is sent to the server. The call will be blocked on the client side until a response has been received from the server that the commit or rollback was executed.

When the commit or rollback is received on the server, it will be committed to the journal, and depending on the value of the parameter **`journal-sync-transactional`** the server will ensure that the commit or rollback is durably persisted to storage before sending the response back to the client. If this parameter has the value **`false`** then commit or rollback may not actually get persisted to storage until some time after the response has been sent to the client. In the event of server failure this may mean the commit or rollback never gets persisted to storage. The default value of this parameter is **`true`** so the client can be sure all transaction commits or rollbacks have been persisted to storage by the time the call to commit or rollback returns.

Setting this parameter to **`false`** can improve performance at the expense of some loss of transaction durability.

This parameter is set in **`hornetq-configuration.xml`**

20.2. Guarantees of Non Transactional Message Sends

If messages are sent to a server using a non-transacted session, HornetQ can block sending until the message has definitely reached the server, and a response has been sent back to the client. This can be configured individually for durable and non-durable messages, and is determined by the following two parameters:

`BlockOnDurableSend`
: If this is set to **`true`** then all calls to send for durable messages on non transacted sessions will block until the message has reached the server, and a response has been sent back. The default value is **`true`**.

`BlockOnNonDurableSend`
: If this is set to **`true`** then all calls to send for non-durable messages on non-transacted sessions will be blocked until the message has reached the server, and a response has been sent back. The default value is **`false`**.

Setting the send block to **`true`** can reduce performance since each send requires a network round trip before the next send can be performed. This means the performance of sending messages will be limited by the network round trip time (RTT) of your network, rather than the bandwidth of your network. For better performance it is recommended that you either batch many message sends together in a transaction (since with a transactional session, only the commit/rollback does not block every send), or use the *asynchronous send acknowledgments feature* described in *Section 20.4, "Asynchronous Send Acknowledgments"*.

If you are using JMS and the JMS service on the server to load your JMS connection factory instances into JNDI then these parameters can be configured in **`hornetq-jms.xml`** using the elements **`block-on-durable-send`** and **`block-on-non-durable-send`**. If you're using JMS but not using JNDI then you can set these values directly on the **`HornetQConnectionFactory`** instance using the appropriate setter methods.

If you're using core you can set these values directly on the **`ClientSessionFactory`** instance using the appropriate setter methods.

When the server receives a message sent from a non transactional session, and that message is durable and the message is routed to at least one durable queue, then the server will persist the message in permanent storage. If the journal parameter **`journal-sync-non-transactional`** is set to **`true`** the server will not send a response back to the client until the message has been persisted and the server has a guarantee that the data has been persisted to disk. The default value for this parameter is **`true`**.

20.3. Guarantees of Non-Transactional Acknowledgments

If you are acknowledging the delivery of a message on the client side using a non transacted session, HornetQ can be configured to block the call to acknowledge until the acknowledge has definitely reached the server, and a response has been sent back to the client. This is configured with the parameter **`BlockOnAcknowledge`**. If this is set to **`true`** then all calls to acknowledge on non-transacted sessions will block until the acknowledge has reached the server, and a response has been sent back. You might want to set this to **`true`** if you want to implement a strict *at most once* delivery policy. The default value is **`false`**

20.4. Asynchronous Send Acknowledgments

If you are using a non-transacted session but want a guarantee that every message sent to the server has reached it, then, as discussed in *Section 20.2, "Guarantees of Non Transactional Message Sends"*, you can configure HornetQ to block the call to send until the server has received the message, persisted it and sent back a response. This works well but has a severe performance penalty - each call to send needs to block for at least the time of a network round trip (RTT) - the performance of sending is thus limited by the latency of the network, *not* limited by the network bandwidth.

To remedy this, HornetQ provides a feature called *asynchronous send acknowledgments*. With this feature, HornetQ can be configured to send messages without blocking in one direction and asynchronously getting acknowledgment from the server that the messages were received in a separate stream. By decoupling the send from the acknowledgment of the send, the system is not limited by the network RTT, but is limited by the network bandwidth. Consequently better throughput can be achieved than is possible using a blocking approach, while at the same time having absolute guarantees that messages have successfully reached the server.

The window size for send acknowledgments is determined by the confirmation-window-size parameter on the connection factory or client session factory. Please see *Chapter 34, Client Reconnection and Session Reattachment* for more info on this.

20.4.1. Asynchronous Send Acknowledgments

To use the feature using the core API, implement the interface **`org.hornetq.api.core.client.SendAcknowledgementHandler`** and set a handler instance on your **`ClientSession`**.

Send messages as normal using your **`ClientSession`**, and as messages reach the server, the server will send back an acknowledgment of the send asynchronously. HornetQ calls your handler's **`sendAcknowledged(ClientMessage message)`** method, passing in a reference to the message that was sent.

To enable asynchronous send acknowledgments, make sure **`confirmation-window-size`** is set to a positive integer value. For example, 10MiB

See *Section 11.1.44, "Send Acknowledgments"* for a full working example.

Message Redelivery and Undelivered Messages

Message delivery can be unsuccessful, for example, if the session used to consume a message is rolled back. An undelivered message returns to the queue ready to be redelivered. However, this means multiple unsuccessful deliveries are possible, so messages can remain in the queue, clogging the system.

There are two options for these undelivered messages:

Delayed Redelivery
: Message delivery can be delayed to allow the client time to recover from transient failures and not overload its network or CPU resources.

Dead Letter Address
: Configure a dead letter address, to which messages are sent after being determined undeliverable.

These options can be combined for maximum flexibility.

21.1. Delayed Redelivery

Delaying redelivery can be useful for clients that regularly fail or roll back. Without delayed redelivery, the system can get into a "thrashing" state, where delivery fails and the client rolls back to attempt redelivery ad infinitum, consuming CPU and network resources.

21.1.1. Configuring Delayed Redelivery

Delayed redelivery is defined in the `address-setting` configuration:

```
<!-- delay redelivery of messages for 5s -->
<address-setting match="jms.queue.exampleQueue">
   <redelivery-delay>5000</redelivery-delay>
</address-setting>
```

If a `redelivery-delay` is specified, HornetQ will wait that many milliseconds before redelivering the messages. By default there is no redelivery delay and `redelivery-delay` is set to **`0`**.

Address wildcards can be used to configure redelivery delay for a set of addresses (see *Chapter 13, Understanding the HornetQ Wildcard Syntax*), so redelivery delay does not have to be specified for each address individually.

See *Section 11.1.12, "Delayed Redelivery"* for an example of delayed redelivery configured for use with JMS.

21.2. Dead Letter Addresses

To prevent a client infinitely receiving the same undelivered message (regardless of what is causing the unsuccessful deliveries), messaging systems define *dead letter addresses*: after a specified unsuccessful delivery attempts, the message is removed from the queue and send instead to a dead letter address.

Any such messages can then be diverted to queue(s) where they can later be perused by the system administrator for action to be taken.

HornetQ's addresses can be assigned a dead letter address. Once the messages have be unsuccessfully delivered for a given number of attempts, they are removed from the queue and sent to the dead letter address. These *dead letter* messages can later be consumed for further inspection.

21.2.1. Configuring Dead Letter Addresses

The dead letter address is defined in the `address-setting` configuration:

```
<!-- undelivered messages in exampleQueue will be sent to the dead
letter address deadLetterQueue after 3 unsuccessful delivery attempts-->
<address-setting match="jms.queue.exampleQueue">
   <dead-letter-address>jms.queue.deadLetterQueue</dead-letter-address>
   <max-delivery-attempts>3</max-delivery-attempts>
</address-setting>
```

If a `dead-letter-address` is not specified, messages will be removed after `max-delivery-attempts` unsuccessful attempts.

By default, messages are redelivered a maximum of 10 times. Set `max-delivery-attempts` to **`-1`** for infinite redeliveries.

A dead letter address can be set globally for a set of matching addresses, with `max-delivery-attempts` set to **`-1`** for a specific address setting to allow infinite redeliveries only for this address.

Address wildcards can be used to configure dead letter settings for a set of addresses (see *Chapter 13, Understanding the HornetQ Wildcard Syntax*).

21.2.2. Dead Letter Properties

Dead letter messages which are consumed from a dead letter address have the following property:

`HQ_ORIG_ADDRESS`
: A String property containing the *original address* of the dead letter message.

See *Section 11.1.11, "Dead Letter"* for an example of dead letter address configuration and use with JMS.

21.3. Delivery Count Persistence

In normal use, HornetQ does not persist an updated delivery count until a message is rolled back (that is, the delivery count is not updated *before* the message is delivered to the consumer). In most messaging use cases, the messages are consumed, acknowledged and forgotten as soon as they are consumed. In these cases, updating the delivery count persistently before delivering the message would add an extra persistent step *for each message delivered*, implying a significant performance penalty.

However, if the delivery count is not updated persistently before message delivery, in the event of a server crash, the delivery of some messages may not be reflected in the delivery count. Therefore, during the recovery phase, the server may deliver the message with `redelivered` set to **`false`** when it should be **`true`**.

Since this behavior breaks strict JMS semantics, delivery count can be persisted before message delivery in HornetQ. However, this is disabled by default for performance reasons. To enable this behavior, set `persist-delivery-count-before-delivery` to **`true`** in **`hornetq-configuration.xml`**:

```
<persist-delivery-count-before-delivery>
```

```
  true
</persist-delivery-count-before-delivery>
```

Message Expiry

Messages can be set with an optional *time to live* (TimeToLive) when sending them.

HornetQ will not deliver a message to a consumer after it's time to live has been exceeded. If the message hasn't been delivered before the time to live is reached, the server can discard it.

HornetQ's addresses can be assigned an expiry address so that, when messages are expired, they are removed from the queue and sent to the expiry address. Many different queues can be bound to an expiry address. These *expired* messages can later be consumed for further inspection.

22.1. Message Expiry

Using HornetQ Core API, you can set an expiration time directly on the message:

```
// message will expire in 5000ms from now
message.setExpiration(System.currentTimeMillis() + 5000);
```

JMS MessageProducer allows you to set a TimeToLive for the messages it sent:

```
// messages sent by this producer will be retained for 5s (5000ms) before expiration

producer.setTimeToLive(5000);
```

Expired messages which are consumed from an expiry address have the following properties:

`_HQ_ORIG_ADDRESS`
: a String property containing the *original address* of the expired message

`_HQ_ACTUAL_EXPIRY`
: a Long property containing the *actual expiration time* of the expired message

22.2. Configuring Expiry Addresses

Expiry address are defined in the address-setting configuration:

```
<!-- expired messages in exampleQueue will be sent to the expiry address expiryQueue -->
<address-setting match="jms.queue.exampleQueue">
   <expiry-address>jms.queue.expiryQueue</expiry-address>
</address-setting>
```

If messages are expired and no expiry address is specified, messages are simply removed from the queue and dropped. Address wildcards can be used to configure expiry address for a set of addresses (see *Chapter 13, Understanding the HornetQ Wildcard Syntax*).

22.3. Configuring The Expiry Reaper Thread

A reaper thread will periodically inspect the queues to check if messages have expired.

The reaper thread can be configured with the following properties in **`hornetq-configuration.xml`**

`message-expiry-scan-period`
: How often the queues will be scanned to detect expired messages (in milliseconds, default is 30000ms, set to **`-1`** to disable the reaper thread)

`message-expiry-thread-priority`
The reaper thread priority (it must be between zero and nine; nine being the highest priority. The default is three).

22.4. Example

See *Section 11.1.27, "Message Expiration"* for an example which shows how message expiry is configured and used with JMS.

Large Messages

HornetQ supports sending and receiving of large messages, even when the client and server are running with limited memory. The only realistic limit to the size of a message that can be sent or consumed is the amount of disk space you have available.

To send a large message, the user can set an **`InputStream`** on a message body. When that message is sent, HornetQ will read the **`InputStream`**. For example, a **`FileInputStream`** could be used to send a large message from a large file on disk.

As the **`InputStream`** is read, the data is sent to the server as a stream of fragments. The server persists these fragments to disk as it receives them. When the time comes to deliver them to a consumer they are read back off the disk, also in fragments, and sent down the wire. When the consumer receives a large message it initially receives just the message with an empty body. It can then set an **`OutputStream`** on the message to stream the large message body to a file on disk or elsewhere. At no time is the entire message body stored fully in memory, either on the client or the server.

23.1. Configuring the server

Large messages are stored on a disk directory on the server side, as configured in the main configuration file.

The configuration property **`large-messages-directory`** specifies where large messages are stored.

```
<configuration xmlns="urn:hornetq"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="urn:hornetq /schema/hornetq-configuration.xsd">

...

   <large-message-directory>/data/large-messages</large-message-directory>

...

</configuration>
```

By default the large message directory is **`data/large messages`**

For the best performance it is recommended that the large messages directory is stored on a different physical volume to the message journal or paging directory.

23.2. Setting the limits

Any message larger than a certain size is considered a large message. Large messages will be split up and sent in fragments. This is determined by the parameter **`min-large-message-size`**

The default value is 100KiB.

23.2.1. Using Core API

If the HornetQ Core API is used, the minimal large message size is specified by **`ClientSessionFactory.setMinLargeMessageSize`**.

```
ClientSessionFactory factory =
```

```
  HornetQClient.createClientSessionFactory(new
  TransportConfiguration(NettyConnectorFactory.class.getName()), null);
factory.setMinLargeMessageSize(25 * 1024);
```

Section 16.3, "Configuring the transport directly from the client side" provides more information on how to instantiate the session factory.

23.2.2. Using JMS

If JNDI is used to look up the connection factory, the minimum large message size is specified in **`hornetq-jms.xml`**

```
...
<connection-factory name="ConnectionFactory">
   <connectors>
     <connector-ref connector-name="netty"/>
   </connectors>
   <entries>
      <entry name="ConnectionFactory"/>
      <entry name="XAConnectionFactory"/>
   </entries>
   <min-large-message-size>250000</min-large-message-size>
</connection-factory>
...
```

If the connection factory is being instantiated directly, the minimum large message size is specified by **`HornetQConnectionFactory.setMinLargeMessageSize`**.

23.3. Streaming large messages

HornetQ supports setting the body of messages using input and output streams (**`java.lang.io`**)

These streams are then used directly for sending (input streams) and receiving (output streams) messages.

When receiving messages there are two ways to deal with the output stream; you may choose to block while the output stream is recovered using the method **`ClientMessage.saveOutputStream`** or alternatively using the method **`ClientMessage.setOutputstream`** which will asynchronously write the message to the stream. If you choose the latter the consumer must be kept alive until the message has been fully received.

You can use any kind of stream you like. The most common use case is to send files stored in your disk, but you could also send things such as:

- JDBC Blobs
- **`SocketInputStream`**
- Things recovered from **`HTTPRequests`**, and so on.

Anything that implements **`java.io.InputStream`** for sending messages or **`java.io.OutputStream`** for receiving them can be used.

23.3.1. Streaming over Core API

The following table shows a list of methods available at **`ClientMessage`** which are also available through JMS by the use of object properties.

Table 23.1. org.hornetq.api.core.client.ClientMessage API

Name	Description	JMS Equivalent Property
setBodyInputStream (InputStream)	Set the InputStream used to read a message body when sending it.	JMS_HQ_InputStream
setOutputStream (OutputStream)	Set the OutputStream that will receive the body of a message. This method does not block.	JMS_HQ_OutputStream
saveOutputStream (OutputStream)	Save the body of the message to the **`OutputStream`**. It will block until the entire content is transferred to the **`OutputStream`**.	JMS_HQ_SaveStream

To set the output stream when receiving a core message:

```
...
ClientMessage msg = consumer.receive(...);

// This will block here until the stream was transferred
msg.saveOutputStream(someOutputStream);

ClientMessage msg2 = consumer.receive(...);

// This will not wait the transfer to finish
msg.setOutputStream(someOtherOutputStream);
...
```

Set the input stream when sending a core message:

```
...
ClientMessage msg = session.createMessage();
msg.setInputStream(dataInputStream);
...
```

23.3.2. Streaming over JMS

When using JMS, HornetQ maps the streaming methods on the core API (see *Table 23.1, "org.hornetq.api.core.client.ClientMessage API"*) by setting object properties. You can use the method **`Message.setObjectProperty`** to set the input and output streams.

The **`InputStream`** can be defined through the JMS Object Property JMS_HQ_InputStream on messages being sent:

```
BytesMessage message = session.createBytesMessage();

FileInputStream fileInputStream = new FileInputStream(fileInput);

BufferedInputStream bufferedInput = new BufferedInputStream(fileInputStream);

message.setObjectProperty("JMS_HQ_InputStream", bufferedInput);

someProducer.send(message);
```

The **`OutputStream`** can be set through the JMS Object Property JMS_HQ_SaveStream on messages being received in a blocking way.

```
BytesMessage messageReceived = (BytesMessage)messageConsumer.receive(120000);

File outputFile = new File("huge_message_received.dat");

FileOutputStream fileOutputStream = new FileOutputStream(outputFile);

BufferedOutputStream bufferedOutput = new BufferedOutputStream(fileOutputStream);

// This will block until the entire content is saved on disk
messageReceived.setObjectProperty("JMS_HQ_SaveStream", bufferedOutput);
```

Setting the **`OutputStream`** could also be done in a non-blocking way using the property JMS_HQ_OutputStream.

```
// This will not wait the stream to finish. You need to keep the consumer active.
messageReceived.setObjectProperty("JMS_HQ_OutputStream", bufferedOutput);
```

Note

When using JMS, Streaming large messages are only supported on **`StreamMessage`** and **`BytesMessage`**.

23.4. Streaming Alternative

If you choose not to use the **`InputStream`** or **`OutputStream`** capability of HornetQ, the data can still be accessed directly in an alternative fashion.

On the Core API, get the bytes of the body as described in the *Using Core* chapter.

```
ClientMessage msg = consumer.receive();

byte[] bytes = new byte[1024];
for (int i = 0 ;  i < msg.getBodySize(); i += bytes.length)
{
   msg.getBody().readBytes(bytes);
   // Whatever you want to do with the bytes
}
```

If using JMS API, **`BytesMessage`** and **`StreamMessage`** also supports it transparently.

```
BytesMessage rm = (BytesMessage)cons.receive(10000);

byte data[] = new byte[1024];

for (int i = 0; i < rm.getBodyLength(); i += 1024)
{
   int numberOfBytes = rm.readBytes(data);
   // Do whatever you want with the data
}
```

23.5. Cache Large Messages on client

Large messages are transferred from server to client by streaming. The message is broken into smaller packets and more packets will be received as the message is read. It is because of this that the body of the large message can be read only once, and by consequence a received message can

be sent to another producer only once. The JMS Bridge for instance will not be able to resend a large message in case of failure.

To solve this problem, you can enable the property **`cache-large-message-client`** in the connection factory. If you enable this property the client consumer will create a temporary file to hold the large message content, so it would be possible to resend large messages.

Note

Use this option in the connection factory used by the JMS Bridge if the JMS Bridge is being used for large messages.

23.6. Large message example

Please see *Section 11.1.21, "Large Message"* for an example which shows how large message is configured and used with JMS.

Paging

HornetQ transparently supports huge queues containing millions of messages while the server is running with limited memory.

In such a situation it's not possible to store all of the queues in memory at one time, so HornetQ transparently *pages* messages in and out of memory as they are needed. This allows massive queues with a low memory footprint.

HornetQ will start paging messages to disk when the size of all messages in memory for an address exceeds a configured maximum size.

By default, HornetQ does not page messages; this must be explicitly configured to activate it.

24.1. Page Files

Messages are stored per address on the file system. Each address has an individual folder where messages are stored in multiple files (page files). Each file will contain messages up to a max configured size (**`page-size-bytes`**). When reading page-files all messages on the page-file are read, routed, and the file is deleted as soon as the messages are recovered.

24.2. Configuration

You can configure the location of the paging folder

Global paging parameters are specified on the main configuration file (**`hornetq-configuration.xml`**).

```
<configuration xmlns="urn:hornetq"
               xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
               xsi:schemaLocation="urn:hornetq /schema/hornetq-configuration.xsd">

               ...

               <paging-directory>/somewhere/paging-directory</paging-directory>

               ...
```

`paging-directory`
: This is where page files are stored. HornetQ will create one folder for each address under this configured location. The default for this is data/paging.

24.3. Paging Mode

As soon as messages delivered to an address exceed the configured size, that address alone goes into page mode.

Note

Paging is done individually per address. Configuring a max-size-bytes for an address means each matching address will have a maximum size specified. Please note it *does not* mean that the total overall size of all matching addresses is limited to max-size-bytes.

24.3.1. Configuration

Configuration is done in the address settings, in the main configuration file (**`hornetq-configuration.xml`**).

```
<address-settings>
   <address-setting match="jms.someaddress">
      <max-size-bytes>104857600</max-size-bytes>
      <page-size-bytes>10485760</page-size-bytes>
      <address-full-policy>PAGE</address-full-policy>
   </address-setting>
</address-settings>
```

This is the list of available parameters on the address settings.

Table 24.1. Paging Address Settings

Property Name	Description	Default
`max-size-bytes`	The max memory the address could have before entering on page mode.	-1 (disabled)
`page-size-bytes`	The size of each page file used on the paging system	10MiB (10 * 1024 * 1024 bytes)
`address-full-policy`	This must be set to PAGE for paging to enable. • If the value is PAGE then further messages will be paged to disk. • If the value is DROP then further messages will be silently dropped. • If the value is BLOCK then client message producers will block when they try and send further messages.	PAGE

24.4. Dropping messages

Instead of paging messages when the max size is reached, an address can also be configured to just drop messages when the address is full.

To do this just set the **`address-full-policy`** to **`DROP`** in the address settings

24.5. Blocking producers

Instead of paging messages when the max size is reached, an address can also be configured to block producers from sending further messages when the address is full. This prevents the memory being exhausted on the server.

Producers will automatically unblock and be able to continue sending when memory is freed up on the server.

To do this just set the **`address-full-policy`** to **`BLOCK`** in the address settings.

In the default configuration, all addresses are configured to block producers after 10 MiB of data are in the address.

24.6. Caution with Addresses with Multiple Queues

When a message is routed to an address that has multiple queues bound to it; for example a JMS subscription, there is only one copy of the message in memory. Each queue only deals with a reference to this. This means that the memory is only freed up once all queues referencing the message have delivered it.

For example:

- An address has ten queues
- One of the queues does not deliver its messages (maybe because of a slow consumer).
- Messages continually arrive at the address and paging is started.
- The other nine queues are empty even though messages have been sent.

In this example the process has to wait until the last queue has delivered some of its messages before it can depage and the other queues finally receive some more messages.

Important

Please note that message selectors will only operate on messages in memory. If you have a large amount of messages paged to disk and a selector that only matches some of the paged messages, then those messages will not be consumed until the messages in memory have been consumed.

HornetQ does not scan through page files on disk to locate matching messages. This is not the primary role of a messaging system. A relational database is recommended for implementations using selectors to select small subsets of messages in very large queues, because this functionality is similar to executing queries over tables in a relational database.

Important

Do not set your paging max size to a figure lower than ack-batch-size or your system may appear to hang.

Messages remain in server memory until they are acknowledged the server, therefore contributing to message sizes on a particular address.

If messages are paged to disk for an address, and are being consumed, they will be depaged from disk when enough memory has been freed up in that address after messages have been consumed and acknowledged. However if messages are not acknowledged then more messages will not be depaged since there is no free space in memory. In this case message consumption can appear to hang.

If a message is not acknowledged explicitly, it will be acknowledged according to the "ack-batch-size" setting.

See *Section 11.1.33, “Paging”* for an example which shows how to use paging with HornetQ.

Queue Attributes

Queue attributes can be set in one of two ways; by configuring them using the configuration file or by using the core API. This chapter will explain how to configure each attribute and what effect the attribute has.

25.1. Predefined Queues

Queues can be predefined via configuration at a core level or at a JMS level. First look at a JMS level.

The following shows a queue predefined in the **`hornetq-jms.xml`** configuration file.

```
<queue name="selectorQueue">
   <entry name="/queue/selectorQueue"/>
   <selector string="color='red'"/>
   <durable>true</durable>
</queue>
```

This name attribute of queue defines the name of the queue. When doing this at a JMS level a naming convention is followed so the actual name of the core queue will be **`jms.queue.selectorQueue`**.

The entry element configures the name that will be used to bind the queue to JNDI. This is a mandatory element and the queue can contain multiple of these to bind the same queue to different names.

The selector element defines what JMS message selector the predefined queue will have. Only messages that match the selector will be added to the queue. This is an optional element with a default of null when omitted.

The durable element specifies whether the queue will be persisted. This again is optional and defaults to true if omitted.

Secondly a queue can be predefined at a core level in the **`hornetq-configuration.xml`** file. The following is an example.

```
<queues>
   <queue name="jms.queue.selectorQueue">
      <address>jms.queue.selectorQueue</address>
      <filter string="color='red'"/>
      <durable>true</durable>
</queues>
```

This is very similar to the JMS configuration, but with three real differences which are listed here:

1. The name attribute of queue is the actual name used for the queue with no naming convention as in JMS.
2. The address element defines what address is used for routing messages.
3. There is no entry element.
4. The filter uses the *Core filter syntax* (described in *Chapter 14, Filter Expressions*), *not* the JMS selector syntax.

25.2. Using the API

Queues can also be created using the core API or the management API.

For the core API, queues can be created via the **`org.hornetq.api.core.client.ClientSession`** interface. There are multiple **`createQueue`** methods that support setting all of the previously mentioned attributes. There is one extra attribute that can be set via this API which is **`temporary`**. Setting this to **`true`** means the queue will be deleted once the session is disconnected.

Take a look at *Chapter 30, Management* for a description of the management API for creating queues.

25.3. Configuring Queues Through Address Settings

There are some attributes that are defined against an address wildcard rather than a specific queue. Here is an example of an **`address-setting`** entry that would be found in the **`hornetq-configuration.xml`** file.

```
<address-settings>
   <address-setting match="jms.queue.exampleQueue">
      <dead-letter-address>jms.queue.deadLetterQueue</dead-letter-address>
      <max-delivery-attempts>3</max-delivery-attempts>
      <redelivery-delay>5000</redelivery-delay>
      <expiry-address>jms.queue.expiryQueue</expiry-address>
      <last-value-queue>true</last-value-queue>
      <max-size-bytes>100000</max-size-bytes>
      <page-size-bytes>20000</page-size-bytes>
      <redistribution-delay>0</redistribution-delay>
      <send-to-dla-on-no-route>true</send-to-dla-on-no-route>
      <address-full-policy>PAGE</address-full-policy>
   </address-setting>
</address-settings>
```

Address settings allow you to provide a block of settings which will be applied against any addresses that match the string in the **`match`** attribute. In the above example the settings would only be applied to any addresses which exactly match the address **`jms.queue.exampleQueue`**, but you can also use wildcards to apply sets of configuration against many addresses. The wildcard syntax used is described in *Chapter 13, Understanding the HornetQ Wildcard Syntax*

For example, if you used the **`match`** string **`jms.queue.#`** the settings would be applied to all addresses which start with **`jms.queue.`** which would be all JMS queues.

The meaning of the specific settings are explained fully throughout the user manual, however here is a brief description with a link to the appropriate chapter if available.

`max-delivery-attempts`
: Defines how many times a canceled message can be redelivered before sending it to the **`dead-letter-address`**. A full explanation can be found in *Section 21.2.1, “Configuring Dead Letter Addresses”*.

`redelivery-delay`
: Defines how long to wait before attempting redelivery of a canceled message. Refer to *Section 21.1.1, “Configuring Delayed Redelivery”*.

`expiry-address`
: Defines where to send a message that has expired. Refer to *Section 22.2, “Configuring Expiry Addresses”*.

`last-value-queue`
: Defines whether a queue only uses last values or not. Refer to *Chapter 27, Last-Value Queues*.

`max-size-bytes` and **`page-size-bytes`**
Are used to set paging on an address. This is explained in *Chapter 24, Paging*.

`redistribution-delay`
Defines how long to wait when the last consumer is closed on a queue before redistributing any messages. See *Section 38.6, "Message Redistribution"*.

`send-to-dla-on-no-route`
If a message is sent to an address, but the server does not route it to any queues, (for example, there might be no queues bound to that address, or none of the queues have filters that match) then normally that message would be discarded. However if this parameter is set to true for that address, if the message is not routed to any queues it will instead be sent to the dead letter address (DLA) for that address, if it exists.

`address-full-policy`
This attribute can have one of the following values: PAGE, DROP or BLOCK and determines what happens when an address where **`max-size-bytes`** is specified becomes full. The default value is PAGE. If the value is PAGE then further messages will be paged to disk.

- If the value is DROP then further messages will be silently dropped.
- If the value is BLOCK then client message producers will block when they try and send further messages.

See the following chapters for more info *Chapter 19, Flow Control*, *Chapter 24, Paging*.

Scheduled Messages

Scheduled messages differ from normal messages in that they will not be delivered until a specified time in the future.

To do this, a special property is set on the message before sending it.

26.1. Scheduled Delivery Property

The property name used to identify a scheduled message is **`"_HQ_SCHED_DELIVERY"`** (or the constant **`Message.HDR_SCHEDULED_DELIVERY_TIME`**).

The specified value must be a positive **`long`** corresponding to the time the message must be delivered (in milliseconds). An example of sending a scheduled message using the JMS API is as follows.

```
TextMessage message =
 session.createTextMessage("This is a scheduled message message which will be delivered
   in 5 sec.");
message.setLongProperty("_HQ_SCHED_DELIVERY", System.currentTimeMillis() + 5000);
producer.send(message);

...

// message will not be received immediately but 5 seconds later
TextMessage messageReceived = (TextMessage) consumer.receive();
```

Scheduled messages can also be sent using the core API, by setting the same property on the core message before sending.

26.2. Example

See *Section 11.1.42, "Scheduled Message"* for an example which shows how scheduled messages can be used with JMS.

Last-Value Queues

Last-Value queues are special queues which discard any messages when a newer message with the same value for a well-defined Last-Value property is put in the queue. In other words, a Last-Value queue only retains the last value.

A typical example for Last-Value queue is for stock prices, where you are only interested by the latest value for a particular stock.

27.1. Configuring Last-Value Queues

Last-value queues are defined in the address-setting configuration:

```
<address-setting match="jms.queue.lastValueQueue">
   <last-value-queue>true</last-value-queue>
</address-setting>
```

By default, **`last-value-queue`** is false. Address wildcards can be used to configure Last-Value queues for a set of addresses (see *Chapter 13, Understanding the HornetQ Wildcard Syntax*).

27.2. Using Last-Value Property

The property name used to identify the last value is **`"_HQ_LVQ_NAME"`** (or the constant **`Message.HDR_LAST_VALUE_NAME`** from the Core API).

For example, if two messages with the same value for the Last-Value property are sent to a Last-Value queue, only the latest message will be kept in the queue:

```
// send 1st message with Last-Value property set to STOCK_NAME
TextMessage message =
  session.createTextMessage("1st message with Last-Value property set");
message.setStringProperty("_HQ_LVQ_NAME", "STOCK_NAME");
producer.send(message);

// send 2nd message with Last-Value property set to STOCK_NAME
message =
  session.createTextMessage("2nd message with Last-Value property set");
message.setStringProperty("_HQ_LVQ_NAME", "STOCK_NAME");
producer.send(message);

...

// only the 2nd message will be received: it is the latest with
// the Last-Value property set
TextMessage messageReceived = (TextMessage)messageConsumer.receive(5000);
System.out.format("Received message: %s\n", messageReceived.getText());
```

27.3. Example

See *Section 11.1.22, “Last-Value Queue”* for an example which shows how last value queues are configured and used with JMS.

Message Grouping

Message groups are sets of messages that have the following characteristics:

- Messages in a message group share the same group id; that is, they have the same group identifier property (**`JMSXGroupID`** for JMS, **`_HQ_GROUP_ID`** for HornetQ Core API).
- Messages in a message group are always consumed by the same consumer, even if there are many consumers on a queue. They pin all messages with the same group id to the same consumer. If that consumer closes another consumer is chosen and will receive all messages with the same group id.

Message groups are useful when you want all messages for a certain value of the property to be processed serially by the same consumer.

An example might be orders for a certain stock. You may want orders for any particular stock to be processed serially by the same consumer. To do this you can create a pool of consumers (perhaps one for each stock), then set the stock name as the value of the _HQ_GROUP_ID property.

This will ensure that all messages for a particular stock will always be processed by the same consumer.

28.1. Using Core API

The property name used to identify the message group is **`"_HQ_GROUP_ID"`** (or the constant **`MessageImpl.HDR_GROUP_ID`**). Alternatively, you can set **`autogroup`** to true on the **`SessionFactory`** which will pick a random unique id.

28.2. Using JMS

The property name used to identify the message group is **`JMSXGroupID`**.

```
// send 2 messages in the same group to ensure the same
// consumer will receive both
Message message = ...
message.setStringProperty("JMSXGroupID", "Group-0");
producer.send(message);

message = ...
message.setStringProperty("JMSXGroupID", "Group-0");
producer.send(message);
```

Alternatively, you can set **`autogroup`** to true on the **`HornetQConnectionFactory`** which will pick a random unique id. This can also be set in the **`hornetq-jms.xml`** file like this:

```
<connection-factory name="ConnectionFactory">
   <connectors>
      <connector-ref connector-name="netty-connector"/>
   </connectors>
   <entries>
      <entry name="ConnectionFactory"/>
   </entries>
   <autogroup>true</autogroup>
</connection-factory>
```

Alternatively you can set the group id via the connection factory. All messages sent with producers created via this connection factory will set the **`JMSXGroupID`** to the specified value on all messages

sent. To configure the group id set it on the connection factory in the **hornetq-jms.xml** configuration file as follows:

```
<connection-factory name="ConnectionFactory">
   <connectors>
      <connector-ref connector-name="netty-connector"/>
   </connectors>
   <entries>
      <entry name="ConnectionFactory"/>
   </entries>
   <group-id>Group-0</group-id>
</connection-factory>
```

28.3. Example

See *Section 11.1.28, "Message Group"* for an example which shows how message groups are configured and used with JMS.

28.4. Example

See *Section 11.1.29, "Message Group"* for an example which shows how message groups are configured via a connection factory.

28.5. Clustered Grouping

Using message groups in a cluster is a bit more complex. This is because messages with a particular group id can arrive on any node so each node needs to know which group ids are bound to which consumer on which node. The consumer handling messages for a particular group id may be on a different node of the cluster, so each node needs to know this information so it can route the message correctly to the node which has that consumer.

To solve this there is a grouping handler. Each node will have its own grouping handler and when a message is sent with a group id assigned, the handlers will decide between them which route the message should take.

There are two types of handlers; local and remote. Each cluster should choose one node to have a local grouping handler and all the other nodes should have remote handlers. It's the local handler that actually makes the decision as to what route should be used; all the other remote handlers converse with this. Here is a sample configuration for both types of handler. This should be configured in the *hornetq-configuration.xml* file.

```
<grouping-handler name="my-grouping-handler">
   <type>LOCAL</type>
   <address>jms</address>
   <timeout>5000</timeout>
</grouping-handler>
<grouping-handler name="my-grouping-handler">
   <type>REMOTE</type>
   <address>jms</address>
   <timeout>5000</timeout>
</grouping-handler>
```

The *address* attribute refers to a cluster connection and the address it uses. Please refer to the clustering section on how to configure clusters. The *timeout* attribute refers to how long to wait for a decision to be made; an exception will be thrown during the send if this timeout is reached. This ensures that strict ordering is kept.

The decision about where a message should be routed is initially proposed by the node that receives the message. The node will pick a suitable route as per the normal clustered routing conditions (that is, circulate available queues, use a local queue first, and choose a queue that has a consumer). If the proposal is accepted by the grouping handlers, the node will route messages to this queue from that point on. If rejected an alternative route will be offered and the node will route to that queue indefinitely. All other nodes will also route to the queue chosen at proposal time. Once the message arrives at the queue then normal single server message group semantics take over and the message is pinned to a consumer on that queue.

You may have noticed that there is a single point of failure with the single local handler. If this node crashes then no decisions will be able to be made. Any messages sent will be not be delivered and an exception will be thrown. To avoid this happening, local handlers can be replicated on another backup node. Simple create your back up node and configure it with the same local handler.

28.5.1. Clustered Grouping Best Practices

Some best practices should be followed when using clustered grouping:

1. Make sure your consumers are distributed evenly across the different nodes if possible. This is only an issue if you are creating and closing consumers regularly. Since messages are always routed to the same queue once pinned, removing a consumer from this queue may leave it with no consumers, meaning the queue will just keep receiving the messages. Avoid closing consumers or make sure that you always have plenty of consumers, that is, if you have three nodes, have three consumers.

2. Use durable queues if possible. If queues are removed once a group is bound to it, then it is possible that other nodes may still try to route messages to it. This can be avoided by making sure that the queue is deleted by the session that is sending the messages. This means that when the next message is sent it is sent to the node where the queue was deleted meaning a new proposal can successfully take place. Alternatively you could just start using a different group id.

3. Always make sure that the node that has the local grouping handler is replicated. These means that grouping can still occur on failover.

28.5.2. Clustered Grouping Example

See *Section 11.1.6, “Clustered Grouping”* for an example of how to configure message groups with a HornetQ cluster.

Pre-Acknowledge Mode

JMS specifies three acknowledgment modes:

- **`AUTO_ACKNOWLEDGE`**
- **`CLIENT_ACKNOWLEDGE`**
- **`DUPS_OK_ACKNOWLEDGE`**

However there is another case which is not supported by JMS. In some cases you can afford to lose messages in event of failure, so it would make sense to acknowledge the message on the server *before* delivering it to the client.

This extra mode is supported by HornetQ and will call it *pre-acknowledge* mode.

The disadvantage of acknowledging on the server before delivery is that the message will be lost if the system crashes *after* acknowledging the message on the server but *before* it is delivered to the client. In that case, the message is lost and will not be recovered when the system restarts.

Depending on your messaging case, **`pre-acknowledgement`** mode can avoid extra network traffic and CPU at the cost of coping with message loss.

An example of a use case for pre-acknowledgment is for stock price update messages. With these messages it might be reasonable to lose a message in event of crash, since the next price update message will arrive soon, overriding the previous price.

Note

Please note, that if you use pre-acknowledge mode, then you will lose transactional semantics for messages being consumed. This is because they are being acknowledged first on the server, not when you commit the transaction.

29.1. Using PRE_ACKNOWLEDGE

This can be configured in the **`hornetq-jms.xml`** file on the **`connection factory`** like this:

```
<connection-factory name="ConnectionFactory">
   <connectors>
      <connector-ref connector-name="netty-connector"/>
   </connectors>
   <entries>
      <entry name="ConnectionFactory"/>
   </entries>
   <pre-acknowledge>true</pre-acknowledge>
</connection-factory>
```

Alternatively, to use pre-acknowledgment mode using the JMS API, create a JMS Session with the **`HornetQSession.PRE_ACKNOWLEDGE`** constant.

```
// messages will be acknowledge on the server *before* being delivered to the client
Session session = connection.createSession(false, HornetQSession.PRE_ACKNOWLEDGE);
```

Or you can set pre-acknowledge directly on the **`HornetQConnectionFactory`** instance using the setter method.

To use pre-acknowledgment mode using the core API you can set it directly on the `ClientSessionFactory` instance using the setter method.

29.2. Example

See *Section 11.1.34, "Pre-Acknowledge"* for an example which shows how to use pre-acknowledgment mode with JMS.

Management

HornetQ has an extensive management API that allows a user to:

- Modify a server configuration;
- Create new resources (for example, JMS queues and topics);
- Inspect these resources (for example, how many messages are currently held in a queue);
- Interact with it (that is, to remove messages from a queue).

All the operations allow a client to *manage* HornetQ. It also allows clients to subscribe to management notifications.

There are three ways to manage HornetQ:

- Using JMX; JMX is the standard way to manage Java applications.
- Using the core API; management operations are sent to HornetQ server using core messages.
- Using the JMS API; management operations are sent to HornetQ server using JMS messages.

Although there are three different ways to manage HornetQ, each API supports the same functionality. If it is possible to manage a resource using JMX it is also possible to achieve the same result using core messages or JMS messages.

This choice depends on your requirements, your application settings, and your environment, to decide which way suits you best.

30.1. The Management API

The management API is the same regardless of the way you invoke management operations. A Java interface describing what can be invoked exists for each type of managed resource.

HornetQ exposes its managed resources in two packages:

- Core resources are located in the *org.hornetq.api.core.management* package.
- *JMS* resources are located in the *org.hornetq.api.jms.management* package.

The way to invoke a management operation depends whether JMX, core messages, or JMS messages are used.

Note

A few management operations requires a filter parameter to choose which messages are involved by the operation. Passing **`null`** or an empty string means that the management operation will be performed on *all messages*.

30.1.1. Core Management API

HornetQ defines a core management API to manage core resources. In summary:

30.1.1.1. Core Server Management

Listing, creating, deploying and destroying queues

A list of deployed core queues can be retrieved using the `getQueueNames()` method.

Core queues can be created or destroyed using the management operations:

- `createQueue()`
- `deployQueue()`
- `destroyQueue()` on the **`HornetQServerControl`** (with the ObjectName **`org.hornetq:module=Core,type=Server`** or the resource name **`core.server`**)

`createQueue` will fail if the queue already exists while `deployQueue` will do nothing.

Pausing and resuming Queues

The `QueueControl` can pause and resume the underlying queue. When a queue is paused, it will receive messages but will not deliver them. When it's resumed, it will begin delivering the queued messages.

Listing and closing remote connections

Client remote addresses can be retrieved using `listRemoteAddresses()`. It is also possible to close the connections associated with a remote address using the `closeConnectionsForAddress()` method.

Alternatively, connection ids can be listed using `listConnectionIDs()` and all the sessions for a given connection id can be listed using `listSessions()`.

Transaction heuristic operations

In the case of a server crash, some transactions may require manual intervention when the server restarts. The `listPreparedTransactions()` method lists the transactions which are in the prepared states (the transactions are represented as opaque Base64 Strings.) To commit or rollback a given prepared transaction, the `commitPreparedTransaction()` or `rollbackPreparedTransaction()` method can be used to resolve heuristic transactions. Heuristically completed transactions can be listed using the `listHeuristicCommittedTransactions()` and `listHeuristicRolledBackTransactions` methods.

Enabling and resetting Message counters

Message counters can be enabled or disabled using the `enableMessageCounters()` or `disableMessageCounters()` method. To reset message counters, it is possible to invoke `resetAllMessageCounters()` and `resetAllMessageCounterHistories()` methods.

Retrieving the server configuration and attributes

The `HornetQServerControl` exposes HornetQ server configuration through all its attributes (for example, `getVersion()` method to retrieve the server's version, and so on.)

30.1.1.2. Core Address Management

Core addresses can be managed using the **`AddressControl`** class (with the ObjectName **`org.hornetq:module=Core,type=Address,name="<the address name>"`** or the resource name **`core.address.<the address name>`**).

Modifying roles and permissions for an address

You can add or remove roles associated to a queue using the `addRole()` or `removeRole()` methods. You can list all the roles associated with the queue with the `getRoles()` method.

30.1.1.3. Core Queue Management

The bulk of the core management APIs deal with core queues. The **`QueueControl`** class defines the core queue management operations (with the ObjectName **`org.hornetq:module=Core,type=Queue,address="<the bound address>",name="<the queue name>"`** or the resource name **`core.queue.<the queue name>`**).

Most of the management operations on queues take either a single message id (for example, to remove a single message) or a filter (for example, to expire all messages with a given property.)

Expiring, sending to a dead letter address and moving messages

Messages can be expired from a queue by using the `expireMessages()` method. If an expiry address is defined, messages will be sent to it, otherwise they are discarded. The queue's expiry address can be set with the `setExpiryAddress()` method.

Messages can also be sent to a dead letter address with the `sendMessagesToDeadLetterAddress()` method. It returns the number of messages which are sent to the dead letter address. If a dead letter address is not defined, message are removed from the queue and discarded. The queue's dead letter address can be set with the `setDeadLetterAddress()` method.

Messages can also be moved from a queue to another queue by using the `moveMessages()` method.

Listing and removing messages

Messages can be listed from a queue by using the `listMessages()` method which returns an array of Map; one Map for each message.

Messages can also be removed from the queue by using the `removeMessages()` method which returns a boolean for the single message id variant or the number of removed messages for the filter variant. The `removeMessages()` method takes a filter argument to remove only filtered messages. Setting the filter to an empty string will remove all messages.

Counting messages

The number of messages in a queue is returned by the `getMessageCount()` method. Alternatively, the `countMessages()` will return the number of messages in the queue which match a given filter

Changing message priority

The message priority can be changed by using the `changeMessagesPriority()` method which returns a boolean for the single message id variant or the number of updated messages for the filter variant.

Message counters

Message counters can be listed for a queue with the `listMessageCounter()` and `listMessageCounterHistory()` methods (see *Section 30.6, "Message Counters"*). The message counters can also be reset for a single queue using the `resetMessageCounter()` method.

Retrieving the queue attributes

The `QueueControl` exposes core queue settings through its attributes (for example, `getFilter()` to retrieve the queue's filter if it was created with one, `isDurable()` to know whether the queue is durable or not, and so on).

Pausing and resuming Queues
: The `QueueControl` can pause and resume the underlying queue. When a queue is paused, it will receive messages but will not deliver them. When it is resumed, it will begin delivering the queued messages.

30.1.1.4. Other Core Resources Management

HornetQ allows the user to start and stop its remote resources (acceptors, diverts, bridges, and so on) so that a server can be taken offline without stopping it completely (for example, if other management operations must be performed such as resolving heuristic transactions). These resources are:

Acceptors
: Acceptors can be started or stopped using the `start()` or. `stop()` method on the **`AcceptorControl`** class (with the ObjectName **`org.hornetq:module=Core,type=Acceptor,name="<the acceptor name>"`** or the resource name **`core.acceptor.<the address name>`**). The parameters of the acceptors can be retrieved using the **`AcceptorControl`** attributes (see *Section 16.1, "Understanding Acceptors"*)

Diverts
: Diverts can be started or stopped using the `start()` or `stop()` method on the **`DivertControl`** class (with the ObjectName **`org.hornetq:module=Core,type=Divert,name=<the divert name>`** or the resource name **`core.divert.<the divert name>`**). Diverts parameters can be retrieved using the **`DivertControl`** attributes (see *Chapter 35, Diverting and Splitting Message Flows*)

Bridges
: They can be started or stopped using the `start()` (resp. `stop()`) method on the **`BridgeControl`** class (with the ObjectName **`org.hornetq:module=Core,type=Bridge,name="<the bridge name>"`** or the resource name **`core.bridge.<the bridge name>`**). Bridges parameters can be retrieved using the **`BridgeControl`** attributes (see *Chapter 36, Core Bridges*)

Broadcast groups
: Broadcast groups can be started or stopped using the `start()` or `stop()` method on the **`BroadcastGroupControl`** class (with the ObjectName **`org.hornetq:module=Core,type=BroadcastGroup,name="<the broadcast group name>"`** or the resource name **`core.broadcastgroup.<the broadcast group name>`**). Broadcast groups parameters can be retrieved using the **`BroadcastGroupControl`** attributes (see *Section 38.2.1, "Broadcast Groups"*)

Discovery groups
: They can be started or stopped using the `start()` or `stop()` method on the **`DiscoveryGroupControl`** class (with the ObjectName **`org.hornetq:module=Core,type=DiscoveryGroup,name="<the discovery group name>"`** or the resource name **`core.discovery.<the discovery group name>`**). Discovery groups parameters can be retrieved using the **`DiscoveryGroupControl`** attributes (see *Section 38.2.2, "Discovery Groups"*)

Cluster connections
: They can be started or stopped using the `start()` or `stop()` method on the **`ClusterConnectionControl`** class (with the ObjectName **`org.hornetq:module=Core,type=ClusterConnection,name="<the cluster connection name>"`** or the resource name **`core.clusterconnection.<the cluster connection name>`**). Cluster connections parameters can be retrieved using

the **ClusterConnectionControl** attributes (see *Section 38.3.1, "Configuring Cluster Connections"*)

30.1.2. JMS Management API

HornetQ defines a JMS Management API to manage JMS administrated objects (that is, JMS queues, topics and connection factories).

30.1.2.1. JMS Server Management

JMS Resources (connection factories and destinations) can be created using the **JMSServerControl** class (with the ObjectName **org.hornetq:module=JMS,type=Server** or the resource name **jms.server**).

Listing, creating, destroying connection factories

Names of the deployed connection factories can be retrieved by the `getConnectionFactoryNames()` method.

JMS connection factories can be created or destroyed using the `createConnectionFactory()` methods or `destroyConnectionFactory()` methods. These connection factories are bound to JNDI so that JMS clients can look them up. If a graphical console is used to create the connection factories, the transport parameters are specified in the text field input as a comma-separated list of key=value (that is, **key1=10, key2="value", key3=false**). If there are multiple transports defined, you need to enclose the key/value pairs between curly braces. For example **{key=10}, {key=20}**. In that case, the first **key** will be associated to the first transport configuration and the second **key** will be associated to the second transport configuration (see *Chapter 16, Configuring the Transport* for a list of the transport parameters)

Listing, creating, destroying queues

Names of the deployed JMS queues can be retrieved by the `getQueueNames()` method.

JMS queues can be created or destroyed using the `createQueue()` methods or `destroyQueue()` methods. These queues are bound to JNDI so that JMS clients can look them up

Listing, creating/destroying topics

Names of the deployed topics can be retrieved by the `getTopicNames()` method.

JMS topics can be created or destroyed using the `createTopic()` or `destroyTopic()` methods. These topics are bound to JNDI so that JMS clients can look them up.

Listing and closing remote connections

JMS Clients remote addresses can be retrieved using `listRemoteAddresses()`. It is also possible to close the connections associated with a remote address using the `closeConnectionsForAddress()` method.

Alternatively, connection ids can be listed using `listConnectionIDs()` and all the sessions for a given connection id can be listed using `listSessions()`.

30.1.2.2. JMS ConnectionFactory Management

JMS Connection Factories can be managed using the **ConnectionFactoryControl** class (with the ObjectName **org.hornetq:module=JMS,type=ConnectionFactory,name="<the connection factory name>"** or the resource name **jms.connectionfactory.<the connection factory name>**).

Retrieving connection factory attributes
: The `ConnectionFactoryControl` exposes JMS ConnectionFactory configuration through its attributes (that is, `getConsumerWindowSize()` to retrieve the consumer window size for flow control, `isBlockOnNonDurableSend()` to know whether the producers created from the connection factory will block when sending non-durable messages, and so on).

30.1.2.3. JMS Queue Management

JMS queues can be managed using the **`JMSQueueControl`** class (with the ObjectName **`org.hornetq:module=JMS,type=Queue,name="<the queue name>"`** or the resource name **`jms.queue.<the queue name>`**).

The management operations on a JMS queue are very similar to the operations on a core queue.

Expiring, sending to a dead letter address and moving messages
: Messages can be expired from a queue by using the `expireMessages()` method. If an expiry address is defined, messages will be sent to it, otherwise they are discarded. The queue's expiry address can be set with the `setExpiryAddress()` method.

  Messages can also be sent to a dead letter address with the `sendMessagesToDeadLetterAddress()` method. It returns the number of messages which are sent to the dead letter address. If a dead letter address is not defined, message are removed from the queue and discarded. The queue's dead letter address can be set with the `setDeadLetterAddress()` method.

  Messages can also be moved from a queue to another queue by using the `moveMessages()` method.

Listing and removing messages
: Messages can be listed from a queue by using the `listMessages()` method which returns an array of Map, one Map for each message.

  Messages can also be removed from the queue by using the `removeMessages()` method which returns a boolean for the single message ID variant or the number of removed messages for the filter variant. The `removeMessages()` method takes a `filter` argument to remove only filtered messages. Setting the filter to an empty string will in effect remove all messages.

Counting messages
: The number of messages in a queue is returned by the `getMessageCount()` method. Alternatively, the `countMessages()` will return the number of messages in the queue which match a given filter

Changing message priority
: The message priority can be changed by using the `changeMessagesPriority()` method which returns a boolean for the single message id variant or the number of updated messages for the filter variant.

Message counters
: Message counters can be listed for a queue with the `listMessageCounter()` and `listMessageCounterHistory()` methods (see *Section 30.6, “Message Counters”*)

Retrieving the queue attributes
: The `JMSQueueControl` exposes JMS queue settings through its attributes (for example, **`isTemporary()`** to know whether the queue is temporary or not, **`isDurable()`** to know whether the queue is durable or not, and so on.)

Pausing and resuming queues
: The `JMSQueueControl` can pause and resume the underlying queue. When the queue is paused it will continue to receive messages but will not deliver them. When resumed again it will deliver the enqueued messages.

30.1.2.4. JMS Topic Management

JMS Topics can be managed using the **`TopicControl`** class (with the ObjectName **`org.hornetq:module=JMS,type=Topic,name="<the topic name>"`** or the resource name **`jms.topic.<the topic name>`**).

Listing subscriptions and messages
: JMS topics subscriptions can be listed using the `listAllSubscriptions()`, `listDurableSubscriptions()`, `listNonDurableSubscriptions()` methods. These methods return arrays of Object representing the subscriptions information (subscription name, client ID, durability, message count, and so on). It is also possible to list the JMS messages for a given subscription with the `listMessagesForSubscription()` method.

Dropping subscriptions
: Durable subscriptions can be dropped from the topic using the `dropDurableSubscription()` method.

Counting subscriptions messages
: The `countMessagesForSubscription()` method can be used to determine the number of messages held for a given subscription (with an optional message selector to determine the number of messages matching the selector).

30.2. Using Management Via JMX

HornetQ can be managed using JMX.

The management API is exposed by HornetQ using MBeans interfaces. HornetQ registers its resources with the domain **`org.hornetq`**.

For example, the **`ObjectName`** to manage a JMS Queue **`exampleQueue`** is:

```
org.hornetq:module=JMS,type=Queue,name="exampleQueue"
```

The MBean is:

```
org.hornetq.api.jms.management.JMSQueueControl
```

The MBean's **`ObjectName`** is built using the helper class **`org.hornetq.api.core.management.ObjectNameBuilder`**. You can also use **`jconsole`** to find the **`ObjectName`** of the MBeans you want to manage.

Managing HornetQ using JMX is identical to management of any Java Applications using JMX. It can be done by reflection or by creating proxies of the MBeans.

30.2.1. Configuring JMX

By default, JMX is enabled to manage HornetQ. It can be disabled by setting `jmx-management-enabled` to **`false`** in **`hornetq-configuration.xml`**:

```
<!-- false to disable JMX management for HornetQ -->
<jmx-management-enabled>false</jmx-management-enabled>
```

If JMX is enabled, HornetQ can be managed locally using **`jconsole`**.

Note

Remote connections to JMX are not enabled by default for security reasons. System properties must be set in **`run.sh`** or **`run.bat`** scripts.

By default, the HornetQ server uses the JMX domain "org.hornetq". To manage several HornetQ servers from the same MBeanServer, the JMX domain can be configured for each individual HornetQ server by setting **`jmx-domain`** in **`hornetq-configuration.xml`**:

```
<!-- use a specific JMX domain for HornetQ MBeans -->
<jmx-domain>my.org.hornetq</jmx-domain>
```

30.2.2. Example

See *Section 11.1.20, "JMX Management"* for an example which shows how to use a remote connection to JMX and MBean proxies to manage HornetQ.

30.3. Using Management Via Core API

The core management API in HornetQ is called by sending core messages to the *management address*.

Management messages are regular core messages with well-known properties that the server needs to understand to interact with the management API:

- The name of the managed resource
- The name of the management operation
- The parameters of the management operation

When a management message is sent to the management address, the HornetQ server will handle it, extract the information, invoke the operation on the managed resources and send a *management reply* to the management message's reply-to address (specified by **`ClientMessageImpl.REPLYTO_HEADER_NAME`**).

A **`ClientConsumer`** can be used to consume the management reply and retrieve the result of the operation (if any) stored in the body of the reply. For portability, results are returned as a JSON string rather than Java Serialization. The **`org.hornetq.api.core.management.ManagementHelper`** can be used to convert the JSON string to Java objects.

These steps can be simplified to make it easier to invoke management operations using Core messages:

Procedure 30.1. Invoking Management Operations

1. **Step One**
   Create a **`ClientRequestor`** to send messages to the management address and receive replies

2. **Step Two**
   Create a **`ClientMessage`**

3. **Step Three**
   Use the helper class **`org.hornetq.api.core.management.ManagementHelper`** to fill the message with the management properties.

4. **Step Four**
   Send the message using the **`ClientRequestor`**

5. **Step Five**
   Use the helper class **`org.hornetq.api.core.management.ManagementHelper`** to retrieve the operation result from the management reply.

For example, to find out the number of messages in the core queue **`exampleQueue`**:

```
ClientSession session = ...
ClientRequestor requestor = new ClientRequestor(session, "jms.queue.hornetq.management");
ClientMessage message = session.createMessage(false);
ManagementHelper.putAttribute(message, "core.queue.exampleQueue", "messageCount");
ClientMessage reply = requestor.request(m);
int count = (Integer) ManagementHelper.getResult(reply);
System.out.println("There are " + count + " messages in exampleQueue");
```

Management operation name and parameters must conform to the Java interfaces defined in the **`management`** packages.

Names of the resources are built using the helper class **`org.hornetq.api.core.management.ResourceNames`** and are straightforward (**`core.queue.exampleQueue`** for the Core Queue **`exampleQueue`**, **`jms.topic.exampleTopic`** for the JMS Topic **`exampleTopic`**, and so on).

30.3.1. Configuring Core Management

The management address to send management messages is configured in **`hornetq-configuration.xml`**:

```
<management-address>jms.queue.hornetq.management</management-address>
```

By default, the address is **`jms.queue.hornetq.management`** (it is prepended by "jms.queue" so that JMS clients can also send management messages).

The management address requires a *special* user permission **`manage`** to be able to receive and handle management messages. This is also configured in **`hornetq-configuration.xml`**:

```
<!-- users with the admin role will be allowed to manage -->
<!-- HornetQ using management messages        -->
<security-setting match="jms.queue.hornetq.management">
   <permission type="manage" roles="admin" />
</security-setting>
```

30.4. Using Management Via JMS

Using JMS messages to manage HornetQ is very similar to using core API.

An important difference is that JMS requires a JMS queue to send the messages to (instead of an address for the core API).

The *management queue* is a special queue and needs to be instantiated directly by the client:

```
Queue managementQueue = HornetQJMSClient.createQueue("hornetq.management");
```

All the other steps are the same than for the Core API but they use JMS API instead:

1. Create a **`QueueRequestor`** to send messages to the management address and receive replies.
2. Create a **`Message.`**
3. Use the helper class **`org.hornetq.api.jms.management.JMSManagementHelper`** to fill the message with the management properties.
4. Send the message using the **`QueueRequestor`**.
5. Use the helper class **`org.hornetq.api.jms.management.JMSManagementHelper`** to retrieve the operation result from the management reply.

For example, to know the number of messages in the JMS queue **`exampleQueue`**:

```
Queue managementQueue = HornetQJMSClient.createQueue("hornetq.management");

QueueSession session = ...
QueueRequestor requestor = new QueueRequestor(session, managementQueue);
connection.start();
Message message = session.createMessage();
JMSManagementHelper.putAttribute(message, "jms.queue.exampleQueue", "messageCount");
Message reply = requestor.request(message);
int count = (Integer)JMSManagementHelper.getResult(reply);
System.out.println("There are " + count + " messages in exampleQueue");
```

30.4.1. Configuring JMS Management

Whether JMS or the core API is used for management, the configuration steps are the same (see *Section 30.3.1, "Configuring Core Management"*).

30.4.2. Example

See *Section 11.1.24, "Management"* for an example which shows how to use JMS messages to manage the HornetQ server.

30.5. Management Notifications

HornetQ emits *notifications* to inform listeners of potentially interesting events (creation of new resources, security violation, and so on).

These notifications can be received three different ways:

- JMX notifications
- Core messages

- JMS messages

30.5.1. JMX Notifications

If JMX is enabled (see *Section 30.2.1, "Configuring JMX"*), JMX notifications can be received by subscribing to two MBeans:

- **`org.hornetq:module=Core,type=Server`** for notifications on *core* resources
- **`org.hornetq:module=JMS,type=Server`** for notifications on *JMS* resources

30.5.2. Core Messages Notifications

HornetQ defines a *management notification address*. Core queues can be bound to this address so that clients will receive management notifications as core messages

A core client which wants to receive management notifications must create a core queue bound to the management notification address. It can then receive the notifications from its queue.

Notification messages are regular core messages with additional properties corresponding to the notification (its type, when it occurred, the resources which were concerned, and so on).

Since notifications are regular core messages, it is possible to use message selectors to filter out notifications and receive only a subset of all the notifications emitted by the server.

30.5.2.1. Configuring The Core Management Notification Address

The management notification address to receive management notifications is configured in **`hornetq-configuration.xml`**:

```
<management-notification-address>hornetq.notifications</management-notification-address>
```

By default, the address is **`hornetq.notifications`**.

30.5.3. JMS Messages Notifications

HornetQ's notifications can also be received using JMS messages.

It is similar to receiving notifications using Core API but an important difference is that JMS requires a JMS Destination to receive the messages (preferably a topic).

You must change the server's management notification address to start with **`jms.queue`** (if it is a JMS queue) or **`jms.topic`** (if it is a JMS topic) to use a JMS Destination to receive management notifications.

```
<!-- notifications will be consumed from "notificationsTopic" JMS Topic -->
<management-notification-address>jms.topic.notificationsTopic</management-notification-
address>
```

Once the notification topic is created, you can receive messages from it or set a **`MessageListener`**:

```
Topic notificationsTopic = HornetQJMSClient.createTopic("notificationsTopic");

Session session = ...
```

```
   MessageConsumer notificationConsumer = session.createConsumer(notificationsTopic);
      notificationConsumer.setMessageListener(new MessageListener()
      {
         public void onMessage(Message notif)
         {
            System.out.println("-------------------------");
            System.out.println("Received notification:");
            try
            {
               Enumeration propertyNames = notif.getPropertyNames();
               while (propertyNames.hasMoreElements())
               {
                  String propertyName = (String)propertyNames.nextElement();
                  System.out.format("  %s: %s\n", propertyName,
notif.getObjectProperty(propertyName));
               }
            }
            catch (JMSException e)
            {
            }
            System.out.println("-------------------------");
         }
      });
```

30.5.4. Example

See *Section 11.1.25, "Management Notification"* for an example which shows how to use a JMS **`MessageListener`** to receive management notifications from the HornetQ server.

30.6. Message Counters

Message counters can be used to obtain information on queues over time as HornetQ keeps a history of queue metrics.

They can be used to show trends on queues. For example, using the management API, it would be possible to query the number of messages in a queue at regular intervals. However, this would not be enough to know if the queue is used. The number of messages can remain constant because nobody is sending or receiving messages from the queue or because there are as many messages sent to the queue than messages consumed from it. The number of messages in the queue remains the same in both cases but its use is different.

Message counters gives additional information about the queues:

`count`
: The *total* number of messages added to the queue since the server was started.

`countDelta`
: The number of messages added to the queue *since the last message counter update.*

`depth`
: The *current* number of messages in the queue.

`depthDelta`
: The *overall* number of messages added or removed from the queue *since the last message counter update*. For example, if **`depthDelta`** is equal to **`-10`** this means that overall 10 messages have been removed from the queue.

`lastAddTimestamp`
: The timestamp of the last time a message was added to the queue.

udpateTimestamp
The timestamp of the last message counter update.

30.6.1. Configuring Message Counters

Message counters are disabled by default as they could have a negative effect on memory.

To enable message counters, you can set it to **true** in **hornetq-configuration.xml**:

```
<message-counter-enabled>true</message-counter-enabled>
```

Message counters keeps a history of the queue metrics (10 days by default) and samples all the queues at regular interval (10 seconds by default). If message counters are enabled, these values should be configured to suit your messaging use case in **hornetq-configuration.xml**:

```
<!-- keep history for a week -->
<message-counter-max-day-history>7</message-counter-max-day-history>
<!-- sample the queues every minute (60000ms) -->
<message-counter-sample-period>60000</message-counter-sample-period>
```

Message counters can be retrieved using the Management API. For example, to retrieve message counters on a JMS Queue using JMX:

```

// retrieve a connection to HornetQ's MBeanServer
MBeanServerConnection mbsc = ...
JMSQueueControlMBean queueControl =
 (JMSQueueControl)MBeanServerInvocationHandler.newProxyInstance(mbsc,
   on,
   JMSQueueControl.class,
   false);
// message counters are retrieved as a JSON String

String counters = queueControl.listMessageCounter();
// use the MessageCounterInfo helper class to manipulate message counters more easily
MessageCounterInfo messageCounter = MessageCounterInfo.fromJSON(counters);
System.out.format("%s message(s) in the queue (since last sample: %s)\n",
   counter.getDepth(),
   counter.getDepthDelta());
```

30.6.2. Example

See *Section 11.1.26, “Message Counter”* for an example which shows how to use message counters to retrieve information on a JMS queue.

30.7. Administering HornetQ Resources Using the Admin Console

It is possible to create and configure HornetQ resources via the admin console within the JBoss Enterprise Application Platform.

The admin console will allow you to create destinations (JMS topics and queues) and JMS connection factories.

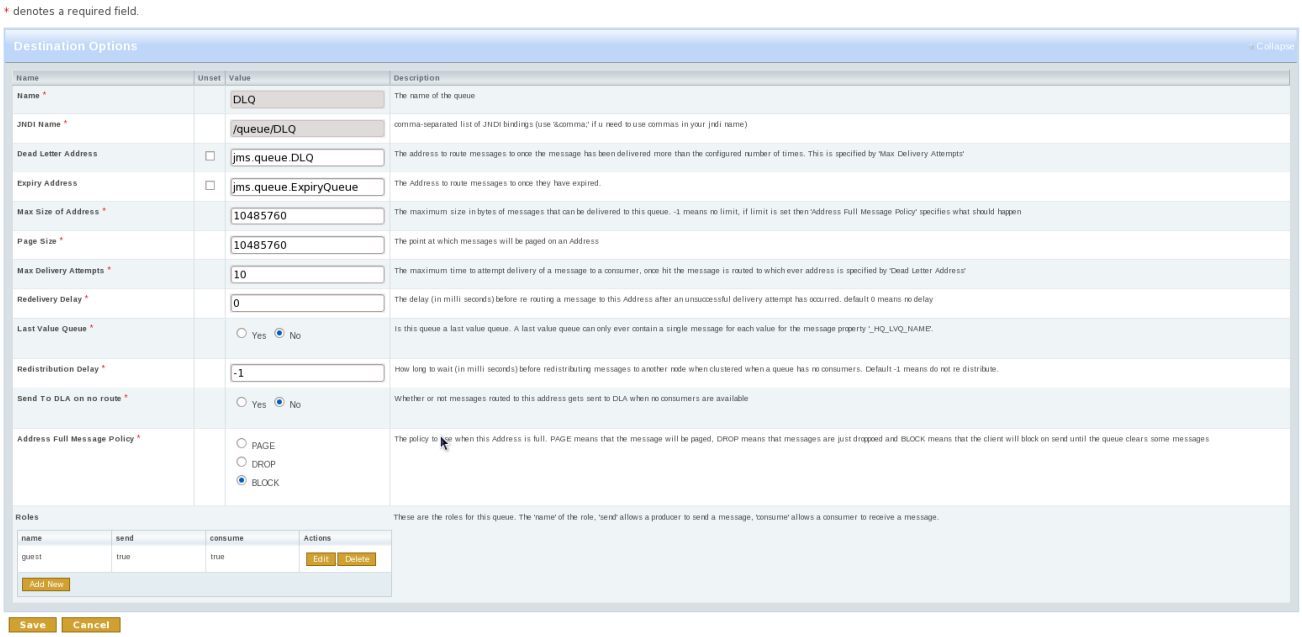
Once logged in to the admin console you will see a JMS Manager item in the left hand tree. All HornetQ resources will be configured via this. This will have child items for JMS queues, topics and

connection factories. Clicking on each node will reveal which resources are currently available. The following sections explain how to create and configure each resource in turn.

30.7.1. JMS Queues

To create a new JMS queue click on the JMS Queues item to reveal the available queues. On the right hand panel you will see an **Add a New Resource** button. Click on this and then choose the default (JMS Queue) template. Click **Continue**. The important things to fill in here are the name of the queue and the JNDI name of the queue. The JNDI name is what you will use to look up the queue in JNDI from your client. For most queues this will be the only info you will need to provide, as sensible defaults are provided for the others. You will also see a security roles section near the bottom. If you do not provide any roles for this queue then the server's default security configuration will be used. After you have created the queue these will be shown in the configuration. All configuration values, except the name and JNDI name, can be changed via the configuration tab after clicking on the queue in the admin console. The following section explains these in more detail.

After highlighting the configuration you will see the following screen:


* denotes a required field.
Destination Options
Collapse
Name
Unset
Value
Description
Name *
DLQ
The name of the queue
JNDI Name *
/queue/DLQ
comma-separated list of JNDI bindings (use 'ɕcomma;' if u need to use commas in your jndi name)
Dead Letter Address
jms.queue.DLQ
The address to route messages to once the message has been delivered more than the configured number of times. This is specified by 'Max Delivery Attempts'
Expiry Address
jms.queue.ExpiryQueue
The Address to route messages to once they have expired.
Max Size of Address *
10485760
The maximum size in bytes of messages that can be delivered to this queue. -1 means no limit, if limit is set then 'Address Full Message Policy' specifies what should happen
Page Size *
10485760
The point at which messages will be paged on an Address
Max Delivery Attempts *
10
The maximum time to attempt delivery of a message to a consumer, once hit the message is routed to which ever address is specified by 'Dead Letter Address'
Redelivery Delay *
0
The delay (in milli seconds) before re routing a message to this Address after an unsuccessful delivery attempt has occurred. default 0 means no delay
Last Value Queue *
Yes
No
Is this queue a last value queue. A last value queue can only ever contain a single message for each value for the message property '_HQ_LVQ_NAME'.
Redistribution Delay *
-1
How long to wait (in milli seconds) before redistributing messages to another node when clustered when a queue has no consumers. Default -1 means do not re distribute.
Send To DLA on no route *
Yes
No
Whether or not messages routed to this address gets sent to DLA when no consumers are available
Address Full Message Policy *
PAGE
DROP
BLOCK
The policy to use when this Address is full. PAGE means that the message will be paged, DROP means that messages are just dropped and BLOCK means that the client will block on send until the queue clears some messages
Roles
These are the roles for this queue. The 'name' of the role, 'send' allows a producer to send a message, 'consume' allows a consumer to receive a message.
name
send
consume
Actions
guest
true
true
Edit
Delete
Add New
Save
Cancel

The name and JNDI name cannot be changed. Recreate the queue with the appropriate settings if you want to change these names. The rest of the configuration options, apart from security roles, relate to address settings for a particular address. The default address settings are picked up from the server configuration. If you change any of these settings or create a queue via the console, a new Address Settings entry will be added. For a full explanation on Address Settings, see *Section 25.3, “Configuring Queues Through Address Settings”*

To delete a queue, click on the **Delete** button beside the queue name in the main JMS queues screen. This will also delete any address settings or security settings previously created for the queue address.

The last part of the configuration options are security roles. If none are provided on creation then the server's default security settings will be shown. If these are changed or updated, new security settings are created for the address of this queue. For more information on security settings, see *Chapter 31, Security*.

It is also possible to view statistics for this queue via the metrics tab. This will show statistics such as message count, consumer count, and so on.

Operations can be performed on a queue via the control tab. This will allow you to start and stop the queue, list, move, expire, and delete messages from the queue and other useful operations. To invoke an operation, click on the button for the operation you want. This will take you to a screen where parameters for the operation can be set. Once set, click the **OK** button to invoke the operation. Results will appear at the bottom of the screen.

30.7.2. JMS Topics

Creating and configuring JMS topics is almost identical to creating queues. The only difference is that the configuration will be applied to the queue representing a subscription.

30.7.3. JMS Connection Factories

The format for creating connection factories is similar to JMS queues and topics. For a list of all the connection factory settings, see the configuration index.

Security

This chapter explores how security works with HornetQ and how it can be configured. To disable security completely, set the **`security-enabled`** property to **`false`** in the **`hornetq-configuration.xml`** file.

For performance reasons security is cached and invalidated periodically. To change this period, set the property **`security-invalidation-interval`**, which is in milliseconds. The default is **`10000`** ms.

31.1. Role based security for addresses

HornetQ contains a flexible role-based security model for applying security to queues, based on their addresses.

As explained in *Chapter 8, Using Core*, HornetQ core primarily consists of sets of queues bound to addresses. A message is sent to an address and the server looks up the set of queues that are bound to that address. The server then routes the message to those sets of queues.

HornetQ allows sets of permissions to be defined against the queues based on their address. An exact match on the address can be used or a wildcard match can be used using the wildcard characters '**#**' and '*****'.

Seven different permissions can be given to the set of queues which match the address. Those permissions are:

`createDurableQueue`
: This permission allows the user to create a durable queue under matching addresses.

`deleteDurableQueue`
: This permission allows the user to delete a durable queue under matching addresses.

`createNonDurableQueue`
: This permission allows the user to create a non-durable queue under matching addresses.

`deleteNonDurableQueue`
: This permission allows the user to delete a non-durable queue under matching addresses.

`send`
: This permission allows the user to send a message to matching addresses.

`consume`
: This permission allows the user to consume a message from a queue bound to matching addresses.

`manage`
: This permission allows the user to invoke management operations by sending management messages to the management address.

For each permission, a list is specified of the roles that are granted that permission. If the user has any of those roles, they will be granted that permission for that set of addresses.

Here is a simple example. Here's a security block from **`hornetq-configuration.xml`** or **`hornetq-queues.xml`** file:

```
<security-setting match="globalqueues.europe.#">
   <permission type="createDurableQueue" roles="admin"/>
```

```
   <permission type="deleteDurableQueue" roles="admin"/>
   <permission type="createNonDurableQueue" roles="admin, guest, europe-users"/>
   <permission type="deleteNonDurableQueue" roles="admin, guest, europe-users"/>
   <permission type="send" roles="admin, europe-users"/>
   <permission type="consume" roles="admin, europe-users"/>
</security-setting>
```

The '#' character signifies "any sequence of words". Words are delimited by the '**.**' character. For a full description of the wildcard syntax please see *Chapter 13, Understanding the HornetQ Wildcard Syntax*. The above security block applies to any address that starts with the string "globalqueues.europe.":

Only users who have the *admin* role can create or delete durable queues bound to an address that starts with the string "globalqueues.europe."

Any users with the roles *admin*, *guest*, or *europe-users* can create or delete temporary queues bound to an address that starts with the string "globalqueues.europe."

Any users with the roles *admin* or *europe-users* can send messages to these addresses or consume messages from queues bound to an address that starts with the string "globalqueues.europe."

The mapping between a user and what roles they have is handled by the security manager. HornetQ ships with a user manager that reads user credentials from a file on disk, and can also plug into JAAS or JBoss Enterprise Application Platform security.

For more information on configuring the security manager, please see *Section 31.4, "Changing the security manager"*.

There can be zero or more **`security-setting`** elements in each xml file. Where more than one match applies to a set of addresses the more specific match takes precedence.

The follow is an example of that. Here's another **`security-setting`** block:

```
<security-setting match="globalqueues.europe.orders.#">
   <permission type="send" roles="europe-users"/>
   <permission type="consume" roles="europe-users"/>
</security-setting>
```

In this **`security-setting`** block the match 'globalqueues.europe.orders.#' is more specific than the previous match 'globalqueues.europe.#'. So any addresses which match 'globalqueues.europe.orders.#' will take their security settings *only* from the latter security-setting block.

Note that settings are not inherited from the former block. All the settings will be taken from the more specific matching block; so for the address 'globalqueues.europe.orders.plastics', the only permissions that exist are **`send`** and **`consume`** for the role europe-users. The permissions **`createDurableQueue`**, **`deleteDurableQueue`**, **`createNonDurableQueue`**, **`deleteNonDurableQueue`** are not inherited from the other security-setting block.

By not inheriting permissions, it allows effective denial of permissions in more specific security-setting blocks by simply not specifying them. Otherwise, it would not be possible to deny permissions in sub-groups of addresses.

31.2. Secure Sockets Layer (SSL) Transport

When messaging clients are connected to servers, or servers are connected to other servers (for example, via bridges) over an untrusted network, HornetQ allows that traffic to be encrypted using the Secure Sockets Layer (SSL) transport.

For more information on configuring the SSL transport, please see *Chapter 16, Configuring the Transport*.

31.3. Basic user credentials

HornetQ ships with a security manager implementation that reads user credentials (user names and passwords), and role information from the **`hornetq-users.properties`** and **`hornetq-users.roles`** files. These files are both located in the **`/conf/props/`** directory within the profile you wish to run.

User credentials, and roles, can easily be added into these files.

Example 31.1, "hornetq-users.properties example file" and *Example 31.2, "hornetq-users.roles example file"*contain four users. Each user is specified in both the .properties and .roles files.

Following the syntax in each file's comments, you assign each user a unique password and attach roles to each user to control what parts of HornetQ they can change.

Example 31.1. hornetq-users.properties example file

```
#
# user=password
#
guest=guest
tim=marmite
andy=doner_kebab
jeff=camembert
```

Example 31.2. hornetq-users.roles example file

```
#
# user=role1,role2,...
#
guest=guest
tim=admin
andy=admin,guest
jeff=europe-users,guest
```

The first thing to note is the guest user defined in both files. A user is classed as a guest when the client does not specify a username/password when creating a session. In this case they will be the user guest and have the role also called *guest*. Multiple roles can be specified for a default user.

We then have three more users: tim, who has the role *admin*; andy, who has the roles *admin* and *guest*; and jeff, who has the roles *europe-users* and *guest*.

31.4. Changing the security manager

If you do not want to use the default security manager, you can specify a different one by editing the file **`hornetq-jboss-beans.xml`** and changing the class for the **`HornetQSecurityManager`** bean.

HornetQ ships with two security manager implementations you can use. One is a JAAS security manager and the other is for integrating with JBoss Enterprise Application Platform security. Alternatively you could write your own implementation by implementing the **`org.hornetq.core.security.SecurityManager`** interface, and specifying the classname of your implementation in the file **`hornetq-jboss-beans.xml`**.

These two implementations are discussed in the next two sections.

31.5. JAAS Security Manager

JAAS (Java Authentication and Authorization Service) is a standard part of the Java platform. It provides a common API for security authentication and authorization, allowing you to plug in your pre-built implementations.

To configure the JAAS security manager to work with your pre-built JAAS infrastructure, you need to specify the security manager as a **`JAASSecurityManager`** in the beans file. Here's an example:

```
<bean name="HornetQSecurityManager"
      class="org.hornetq.integration.jboss.security.JAASSecurityManager">
    <start ignored="true"/>
    <stop ignored="true"/>

    <property name="ConfigurationName">org.hornetq.jms.example.ExampleLoginModule</property>
    <property name="Configuration">
       <inject bean="ExampleConfiguration"/>
    </property>
    <property name="CallbackHandler">
       <inject bean="ExampleCallbackHandler"/>
    </property>
</bean>
```

Note that you need to feed the JAAS security manager with three properties:

ConfigurationName
: The name of the **`LoginModule`** implementation that JAAS must use

Configuration
: The **`Configuration`** implementation used by JAAS

CallbackHandler
: The **`CallbackHandler`** implementation to use if user interaction are required

31.5.1. Example

See *Section 11.1.19, "JAAS"* for an example which shows how HornetQ can be configured to use JAAS.

31.6. JBoss AS Security Manager

The security manager is used when running HornetQ inside the JBoss Enterprise Application Platform. This allows tight integration with the JBoss Enterprise Application Platform's security model.

The class name of this security manager is
`org.hornetq.integration.jboss.security.JBossASSecurityManager`

Take a look at one of the default **`hornetq-jboss-beans.xml`** files for JBoss Enterprise Application Platform that are bundled in the distribution for an example of how this is configured.

31.6.1. Configuring Client Login

JBoss can be configured to allow client login. This is when a Java EE component such as a Servlet or EJB sets security credentials on the current security context and these are used throughout the

call. If you would like these credentials to be used by HornetQ when sending or consuming messages then set **`allowClientLogin`** to **`true`**. This will bypass HornetQ authentication and propagate the provided Security Context. If you would like HornetQ to authenticate using the propagated security then set the `authoriseOnClientLogin` to true also.

Note

If non-blocking messages are sent then there is a chance that these could arrive on the server after the calling thread has completed. This means that the security context has been cleared. If this is the case then messages will need to be sent as blocking messages.

31.7. Changing the username/password for clustering

In order for cluster connections to work correctly, each node in the cluster must make connections to the other nodes. The username/password they use for this should always be changed from the installation default to prevent a security risk.

Please see *Chapter 30, Management* for instructions on how to do this.

Application Server Integration and Java EE

Since HornetQ also provides a JCA adapter, it is also possible to integrate HornetQ as a JMS provider in other Java EE compliant app servers. For instructions on how to integrate a remote JCA adaptor into another application sever, please consult the other application server's instructions.

A JCA Adapter basically controls the inflow of messages to Message-Driven Beans (MDBs) and the outflow of messages sent from other Java EE components, e.g. EJBs and Servlets.

This section explains the basics behind configuring the different Java EE components in the AS.

32.1. Configuring Message-Driven Beans

The delivery of messages to an MDB using HornetQ is configured on the JCA Adapter via a configuration file **`ra.xml`** which can be found under the **`jms-ra.rar`** directory. By default this is configured to consume messages using an InVM connector from the instance of HornetQ running within the application server. The configuration properties are listed later in this chapter.

All MDBs however need to have the destination type and the destination configured. The following example shows how this can be done using annotations:

```
@MessageDriven(name = "MDBExample",
  activationConfig =
    {
      @ActivationConfigProperty
        (propertyName = "destinationType", propertyValue = "javax.jms.Queue"),
      @ActivationConfigProperty
        (propertyName = "destination", propertyValue = "queue/testQueue")
    })
@ResourceAdapter("hornetq-ra.rar")
public class MDBExample implements MessageListener
{
  public void onMessage(Message message)...
}
```

In this example you can see that the MDB will consume messages from a queue that is mapped into JNDI with the binding **`queue/testQueue`**. This queue must be preconfigured in the usual way using the HornetQ configuration files.

The **`ResourceAdapter`** annotation is used to specify which adaptor should be used. To use this you will need to import **`org.jboss.ejb3.annotation.ResourceAdapter`** for JBoss Enterprise Application Platform and later version which can be found in the **`jboss-ejb3-ext-api.jar`** which can be found in the JBoss repository. For JBoss AS 4.X, the annotation to use is **`org.jboss.annotation.ejb.ResourceAdaptor`**.

Alternatively you can add use a deployment descriptor and add something like the following to **`jboss.xml`**

```
<message-driven>
   <ejb-name>ExampleMDB</ejb-name>
   <resource-adapter-name>hornetq-ra.rar</resource-adapter-name>
</message-driven>
```

You can also rename the **`hornetq-ra.rar`** directory to **`jms-ra.rar`** and neither the annotation or the extra descriptor information will be needed. If you do this you will need to edit the **`jms-ds.xml`** datasource file and change **`rar-name`** element.

Note

HornetQ is the default JMS provider for JBoss AS 6. Starting with this AS version, HornetQ resource adapter is named **`jms-ra.rar`** and you no longer need to annotate the MDB for the resource adapter name.

All the examples shipped with the HornetQ distribution use the annotation.

32.1.1. Using Container-Managed Transactions

When an MDB is using Container-Managed Transactions (CMT), the delivery of the message is done within the scope of a JTA transaction. The commit or rollback of this transaction is controlled by the container itself. If the transaction is rolled back then the message delivery semantics will kick in (by default, it will try to redeliver the message up to 10 times before sending to a DLQ). Using annotations this would be configured as follows:

```
@MessageDriven(name = "MDB_CMP_TxRequiredExample",
  activationConfig =
  {
    @ActivationConfigProperty
      (propertyName = "destinationType", propertyValue = "javax.jms.Queue"),
    @ActivationConfigProperty
      (propertyName = "destination", propertyValue = "queue/testQueue")
  })
@TransactionManagement(value= TransactionManagementType.CONTAINER)
@TransactionAttribute(value= TransactionAttributeType.REQUIRED)
@ResourceAdapter("hornetq-ra.rar")
public class MDB_CMP_TxRequiredExample implements MessageListener
{
   public void onMessage(Message message)...
}
```

The **`TransactionManagement`** annotation tells the container to manage the transaction. The **`TransactionAttribute`** annotation tells the container that a JTA transaction is required for this MDB. Note that the only other valid value for this is **`TransactionAttributeType.NOT_SUPPORTED`** which tells the container that this MDB does not support JTA transactions and one should not be created.

It is also possible to inform the container that it must rollback the transaction by calling **`setRollbackOnly`** on the **`MessageDrivenContext`**. The code for this would look something like:

```
@Resource
   MessageDrivenContextContext ctx;

   public void onMessage(Message message)
   {
      try
      {
         //something here fails
      }
      catch (Exception e)
      {
         ctx.setRollbackOnly();
      }
   }
```

If you do not want the overhead of an XA transaction being created every time but you would still like the message delivered within a transaction (i.e. you are only using a JMS resource) then you can configure the MDB to use a local transaction. This would be configured as such:

```
@MessageDriven(name = "MDB_CMP_TxLocalExample",
  activationConfig =
  {
    @ActivationConfigProperty
      (propertyName = "destinationType", propertyValue = "javax.jms.Queue"),
    @ActivationConfigProperty
      (propertyName = "destination", propertyValue = "queue/testQueue"),
    @ActivationConfigProperty
      (propertyName = "useLocalTx", propertyValue = "true")
  })
@TransactionManagement(value = TransactionManagementType.CONTAINER)
@TransactionAttribute(value = TransactionAttributeType.NOT_SUPPORTED)
@ResourceAdapter("hornetq-ra.rar")
public class MDB_CMP_TxLocalExample implements MessageListener
{
   public void onMessage(Message message)...
}
```

32.1.2. Using Bean-Managed Transactions

Message-driven beans can also be configured to use Bean-Managed Transactions (BMT). In this case a User Transaction is created. This would be configured as follows:

```
@MessageDriven(name = "MDB_BMPExample",
  activationConfig =
    {
      @ActivationConfigProperty
        (propertyName = "destinationType", propertyValue = "javax.jms.Queue"),
      @ActivationConfigProperty
        (propertyName = "destination", propertyValue = "queue/testQueue"),
      @ActivationConfigProperty
        (propertyName = "acknowledgeMode", propertyValue = "Dups-ok-acknowledge")
    })
@TransactionManagement(value= TransactionManagementType.BEAN)
@ResourceAdapter("hornetq-ra.rar")
public class MDB_BMPExample implements MessageListener
{
   public void onMessage(Message message)
}
```

When using Bean-Managed Transactions the message delivery to the MDB will occur outside the scope of the user transaction and use the acknowledge mode specified by the user with the **`acknowledgeMode`** property. There are only 2 acceptable values for this **`Auto-acknowledge`** and **`Dups-ok-acknowledge`**. Please note that because the message delivery is outside the scope of the transaction a failure within the MDB will not cause the message to be redelivered.

A user would control the lifecycle of the transaction something like the following:

```
@Resource
  MessageDrivenContext ctx;

  public void onMessage(Message message)
  {
    UserTransaction tx;
    try
    {
      TextMessage textMessage = (TextMessage)message;
      String text = textMessage.getText();
```

```
      UserTransaction tx = ctx.getUserTransaction();
      tx.begin();
      //do some stuff within the transaction
      tx.commit();
    }
    catch (Exception e)
    {
       tx.rollback();
    }
  }
```

32.1.3. Using Message Selectors with Message-Driven Beans

It is also possible to use MDBs with message selectors. To do this simple define your message selector as follows:

```
@MessageDriven(name = "MDBMessageSelectorExample",
  activationConfig =
    {
      @ActivationConfigProperty
        (propertyName = "destinationType", propertyValue = "javax.jms.Queue"),
      @ActivationConfigProperty
        (propertyName = "destination", propertyValue = "queue/testQueue"),
      @ActivationConfigProperty
        (propertyName = "messageSelector", propertyValue = "color = 'RED'")
    })
@TransactionManagement(value= TransactionManagementType.CONTAINER)
@TransactionAttribute(value= TransactionAttributeType.REQUIRED)
@ResourceAdapter("hornetq-ra.rar")
public class MDBMessageSelectorExample implements MessageListener
{
   public void onMessage(Message message)....
}
```

32.2. Sending Messages from within Java EE components

The JCA adapter can also be used for sending messages. The Connection Factory to use is configured by default in the **`jms-ds.xml`** file and is mapped to **`java:/JmsXA`**. Using this from within a Java EE component will mean that the sending of the message will be done as part of the JTA transaction being used by the component.

This means that if the sending of the message fails the overall transaction would rollback and the message be re-sent. Heres an example of this from within an MDB:

```
@MessageDriven(name = "MDBMessageSendTxExample",
  activationConfig =
    {
      @ActivationConfigProperty
        (propertyName = "destinationType", propertyValue = "javax.jms.Queue"),
      @ActivationConfigProperty
        (propertyName = "destination", propertyValue = "queue/testQueue")
    })
@TransactionManagement(value= TransactionManagementType.CONTAINER)
@TransactionAttribute(value= TransactionAttributeType.REQUIRED)
@ResourceAdapter("hornetq-ra.rar")
public class MDBMessageSendTxExample implements MessageListener
{
   @Resource(mappedName = "java:/JmsXA")
   ConnectionFactory connectionFactory;

   @Resource(mappedName = "queue/replyQueue")
   Queue replyQueue;
```

```
   public void onMessage(Message message)
   {
      Connection conn = null;
      try
      {
         //Step 9. We know the client is sending a text message so we cast
         TextMessage textMessage = (TextMessage)message;

         //Step 10. get the text from the message.
         String text = textMessage.getText();

         System.out.println("message " + text);

         conn = connectionFactory.createConnection();

         Session sess = conn.createSession(false, Session.AUTO_ACKNOWLEDGE);

         MessageProducer producer = sess.createProducer(replyQueue);

         producer.send(sess.createTextMessage("this is a reply"));

      }
      catch (Exception e)
      {
         e.printStackTrace();
      }
      finally
      {
         if(conn != null)
         {
            try
            {
               conn.close();
            }
            catch (JMSException e)
            {
            }
         }
      }
   }
   }
```

In JBoss Application Server you can use the JMS JCA adapter for sending messages from EJBs (including Session, Entity and Message-Driven Beans), Servlets (including JSPs), and custom MBeans.

32.3. MDB and Consumer pool size

Most application servers, including JBoss, allow you to configure how many MDBs there are in a pool. In Jboss this is configured via the **`MaxPoolSize`** parameter in the **`ejb3-interceptors-aop.xml`** file. Configuring this has no actual effect on how many sessions/consumers there actually are created. This is because the Resource Adaptor implementation knows nothing about the application servers MDB implementation. So even if you set the MDB pool size to 1, 15 sessions/consumers will be created (this is the default). If you want to limit how many sessions/consumers are created then you need to set the **`maxSession`** parameter either on the resource adapter itself or via an an Activation Config Property on the MDB itself

```
@MessageDriven(name = "MDBMessageSendTxExample",
  activationConfig =
    {
      @ActivationConfigProperty
        (propertyName = "destinationType", propertyValue = "javax.jms.Queue"),
```

```
      @ActivationConfigProperty
        (propertyName = "destination", propertyValue = "queue/testQueue"),
      @ActivationConfigProperty
        (propertyName = "maxSession", propertyValue = "1")
    })
@TransactionManagement(value= TransactionManagementType.CONTAINER)
@TransactionAttribute(value= TransactionAttributeType.REQUIRED)
@ResourceAdapter("hornetq-ra.rar")
public class MyMDB implements MessageListener
{ ....}
```

32.4. Configuring the JCA Adaptor

The Java Connector Architecture (JCA) Adapter is what allows HornetQ to be integrated with Java EE components such as MDBs and EJBs. It configures how components such as MDBs consume messages from the HornetQ server and also how components such as EJBs or Servlets can send messages.

The HornetQ JCA adapter is deployed via the **`jms-ra.rar`** archive. The configuration of the adapter is found in this archive under **`META-INF/ra.xml`**.

The configuration will look something like the following:

```
<resourceadapter>
  <resourceadapter-class>
    org.hornetq.ra.HornetQResourceAdapter
  </resourceadapter-class>
  <config-property>
     <description>The transport type</description>
     <config-property-name>ConnectorClassName</config-property-name>
     <config-property-type>java.lang.String</config-property-type>
     <config-property-value>
       org.hornetq.core.remoting.impl.invm.InVMConnectorFactory
     </config-property-value>
  </config-property>
  <config-property>
     <description>
       The transport configuration. These values must be in the form of key=val;key=val;
     </description>
     <config-property-name>ConnectionParameters</config-property-name>
     <config-property-type>java.lang.String</config-property-type>
     <config-property-value>server-id=0</config-property-value>
  </config-property>

  <outbound-resourceadapter>
     <connection-definition>
        <managedconnectionfactory-class>org.hornetq.ra.HornetQRAManagedConnection
        Factory</managedconnectionfactory-class>

        <config-property>
           <description>The default session type</description>
           <config-property-name>SessionDefaultType</config-property-name>
           <config-property-type>java.lang.String</config-property-type>
           <config-property-value>javax.jms.Queue</config-property-value>
        </config-property>
        <config-property>
           <description>Try to obtain a lock within specified number of seconds; less
           than or equal to 0 disable this functionality</description>
           <config-property-name>UseTryLock</config-property-name>
           <config-property-type>java.lang.Integer</config-property-type>
           <config-property-value>0</config-property-value>
        </config-property>

        <connectionfactory-interface>org.hornetq.ra.HornetQRAConnectionFactory
```

```
        </connectionfactory-interface>
        <connectionfactororg.hornetq.ra.HornetQConnectionFactoryImplonFactoryImpl
        </connectionfactory-impl-class>
        <connection-interface>javax.jms.Session</connection-interface>
        <connection-impl-class>org.hornetq.ra.HornetQRASession
        </connection-impl-class>
    </connection-definition>
    <transaction-support>XATransaction</transaction-support>
    <authentication-mechanism>
        <authentication-mechanism-type>BasicPassword
        </authentication-mechanism-type>
        <credential-interface>javax.resource.spi.security.PasswordCredential
        </credential-interface>
    </authentication-mechanism>
    <reauthentication-support>false</reauthentication-support>
</outbound-resourceadapter>

<inbound-resourceadapter>
    <messageadapter>
        <messagelistener>
            <messagelistener-type>javax.jms.MessageListener</messagelistener-type>
            <activationspec>
                <activationspec-class>org.hornetq.ra.inflow.HornetQActivationSpec
                </activationspec-class>
                <required-config-property>
                    <config-property-name>destination</config-property-name>
                </required-config-property>
            </activationspec>
        </messagelistener>
    </messageadapter>
</inbound-resourceadapter>

</resourceadapter>
```

There are three main parts to this configuration.

1. A set of global properties for the adapter
2. The configuration for the outbound part of the adapter. This is used for creating JMS resources within EE components.
3. The configuration of the inbound part of the adapter. This is used for controlling the consumption of messages via MDBs.

32.4.1. Global Properties

The first element you see is **`resourceadapter-class`** which should be left unchanged. This is the HornetQ resource adapter class.

After that there is a list of configuration properties. This will be where most of the configuration is done. The first two properties configure the transport used by the adapter and the rest configure the connection factory itself.

Note

All connection factory properties will use the defaults if they are not provided, except for the **`reconnectAttempts`** which will default to -1. This signifies that the connection should attempt to reconnect on connection failure indefinitely. This is only used when the adapter is configured to connect to a remote server as an InVM connector can never fail.

The following table explains what each property is for.

Table 32.1. Global Configuration Properties

Property Name	Property Type	Property Description
`ConnectorClassName`	String	The Connector class name (see *Chapter 16, Configuring the Transport* for more information)
`ConnectionParameters`	String	The transport configuration. These parameters must be in the form of **`key1=val1;key2=val2;`** and will be specific to the connector used
`useLocalTx`	boolean	True will enable local transaction optimization.
`UserName`	String	The user name to use when making a connection
`Password`	String	The password to use when making a connection
`BackupConnectorClassName`	String	The backup transport to use in case of failure of the live node
`BackupConnectionParameters`	String	The backup transport configuration parameters
`DiscoveryAddress`	String	The discovery group address to use to autodetect a server
`DiscoveryPort`	Integer	The port to use for discovery
`DiscoveryRefreshTimeout`	Long	The timeout, in milliseconds, to refresh.
`DiscoveryInitialWaitTimeout`	Long	The initial time to wait for discovery.
`ConnectionLoadBalancingPolicyClassName`	String	The load balancing policy class to use.
`ConnectionTTL`	Long	The time to live (in milliseconds) for the connection.
`CallTimeout`	Long	the call timeout (in milliseconds) for each packet sent.
`DupsOKBatchSize`	Integer	the batch size (in bytes) between acknowledgments when using DUPS_OK_ACKNOWLEDGE mode
`TransactionBatchSize`	Integer	the batch size (in bytes) between acknowledgments when using a transactional session

Property Name	Property Type	Property Description
`ConsumerWindowSize`	Integer	the window size (in bytes) for consumer flow control
`ConsumerMaxRate`	Integer	the fastest rate a consumer may consume messages per second
`ConfirmationWindowSize`	Integer	the window size (in bytes) for reattachment confirmations
`ProducerMaxRate`	Integer	the maximum rate of messages per second that can be sent
`MinLargeMessageSize`	Integer	the size (in bytes) before a message is treated as large
`BlockOnAcknowledge`	Boolean	whether or not messages are acknowledged synchronously
`BlockOnNonDurableSend`	Boolean	whether or not non-durable messages are sent synchronously
`BlockOnDurableSend`	Boolean	whether or not durable messages are sent synchronously
`AutoGroup`	Boolean	whether or not message grouping is automatically used
`PreAcknowledge`	Boolean	whether messages are pre acknowledged by the server before sending
`ReconnectAttempts`	Integer	maximum number of retry attempts, default for the resource adapter is -1 (infinite attempts)
`RetryInterval`	Long	the time (in milliseconds) to retry a connection after failing
`RetryIntervalMultiplier`	Double	multiplier to apply to successive retry intervals
`FailoverOnServerShutdown`	Boolean	If true client will reconnect to another server if available
`ClientID`	String	the pre-configured client ID for the connection factory
`ClientFailureCheckPeriod`	Long	the period (in ms) after which the client will consider the connection failed after not receiving packets from the server
`UseGlobalPools`	Boolean	whether or not to use a global thread pool for threads
`ScheduledThreadPoolMaxSize`	Integer	the size of the *scheduled thread* pool

Property Name	Property Type	Property Description
`ThreadPoolMaxSize`	Integer	the size of the thread pool
`SetupAttempts`	Integer	Number of attempts to setup a JMS connection (default is 10, -1 means to attempt infinitely). It is possible that the MDB is deployed before the JMS resources are available. In that case, the resource adapter will try to setup several times until the resources are available. This applies only for inbound connections
`SetupInterval`	Long	Interval in milliseconds between consecutive attempts to setup a JMS connection (default is 2000m). This applies only for inbound connections

32.4.2. Adapter Outbound Configuration

The outbound configuration should remain unchanged as they define connection factories that are used by Java EE components. These Connection Factories can be defined inside a configuration file that matches the name **`*-ds.xml`**. You'll find a default **`jms-ds.xml`** configuration under the **`hornetq`** directory in the JBoss AS deployment. The connection factories defined in this file inherit their properties from the main **`ra.xml`** configuration but can also be overridden. The following example shows how to override them.

Note

Please note that this configuration only applies when HornetQ resource adapter is installed in JBoss Application Server. If you are using another Java EE application server please refer to your application servers documentation for how to do this.

```
<tx-connection-factory>
      <jndi-name>RemoteJmsXA</jndi-name>
      <xa-transaction/>
      <rar-name>jms-ra.rar</rar-name>
      <connection-definition>org.hornetq.ra.HornetQRAConnectionFactory
</connection-definition>
      <config-property name="SessionDefaultType" type="String">javax.jms.Topic
      </config-property>
      <config-property name="ConnectorClassName" type="String">
        org.hornetq.core.remoting.impl.netty.NettyConnectorFactory
      </config-property>
      <config-property name="ConnectionParameters" type="String">
          port=5445</config-property>
      <max-pool-size>20</max-pool-size>
</tx-connection-factory>
```

In this example the connection factory will be bound to JNDI with the name **`RemoteJmsXA`** and can be looked up in the usual way using JNDI or defined within the EJB or MDB as such:

```
@Resource(mappedName="java:/RemoteJmsXA")
private ConnectionFactory connectionFactory;
```

The **`config-property`** elements are what overrides those in the **`ra.xml`** configuration file. Any of the elements pertaining to the connection factory can be overridden here.

The outbound configuration also defines additional properties in addition to the global configuration properties.

Table 32.2. Outbound Configuration Properties

Property Name	Property Type	Property Description
SessionDefaultType	String	the default session type
UseTryLock	Integer	try to obtain a lock within specified number of seconds. less than or equal to 0 disable this functionality

32.4.3. Adapter Inbound Configuration

The inbound configuration should again remain unchanged. This controls what forwards messages onto MDBs. It is possible to override properties on the MDB by adding an activation configuration to the MDB itself. This could be used to configure the MDB to consume from a different server.

The inbound configuration also defines additional properties in addition to the global configuration properties.

Table 32.3. Inbound Configuration Properties

Property Name	Property Type	Property Description
Destination	String	JNDI name of the destination
DestinationType	String	type of the destination, either **`javax.jms.Queue`** or **`javax.jms.Topic`** (default is javax.jms.Queue)
AcknowledgeMode	String	The Acknowledgment mode, either **`Auto-acknowledge`** or **`Dups-ok-acknowledge`** (default is Auto-acknowledge). **`AUTO_ACKNOWLEDGE`** and **`DUPS_OK_ACKNOWLEDGE`** are acceptable values.
MaxSession	Integer	Maximum number of session created by this inbound configuration (default is 15)
MessageSelector	String	the message selector of the consumer
SubscriptionDurability	String	Type of the subscription, either **`Durable`** or **`NonDurable`**
SubscriptionName	String	Name of the subscription
TransactionTimeout	Long	The transaction timeout in milliseconds (default is 0, the transaction does not timeout)

Property Name	Property Type	Property Description
UseJNDI	Boolean	Whether or not use JNDI to look up the destination (default is true)

32.5. Configuring the JBoss Application Server to connect to Remote HornetQ Server

This is a step by step guide on how to configure a JBoss application server that doesn't have HornetQ installed to use a remote instance of HornetQ

32.5.1. Configuring Jboss 5

Download and install JBoss Enterprise Application Platform 5 by following the instructions in the *Installation Guide*, and HornetQ as per the HornetQ installation guide. After that he following steps are required

- Copy the following jars from the HornetQ distribution to the **`lib`** directory of the server profile you have chosen, (for example, **`default`**).
  - hornetq-core-client.jar
  - hornetq-jms-client.jar
  - hornetq-ra.jar (this can be found inside the **`hornetq-ra.rar`** archive)
  - netty.jar
- create the directories **`hornetq-ra.rar`** and **`hornetq-ra.rar/META-INF`** under the **`deploy`** directory in your JBoss config directory
- under the **`hornetq-ra.rar/META-INF`** create a **`ra.xml`** file or copy it from the HornetQ distribution (again it can be found in the **`hornetq-ra.rar`** archive) and configure it as follows

```
                        <?xml version="1.0" encoding="UTF-8"?>

                        <connector xmlns="http://java.sun.com/xml/ns/j2ee"
                                   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                                   xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
                                   http://java.sun.com/xml/ns/j2ee/connector_1_5.xsd"
                                   version="1.5">

                           <description>HornetQ 2.0 Resource Adapter Alternate
Configuration</description>
                           <display-name>HornetQ 2.0 Resource Adapter Alternate
Configuration</display-name>

                           <vendor-name>Red Hat Middleware LLC</vendor-name>
                           <eis-type>JMS 1.1 Server</eis-type>
                           <resourceadapter-version>1.0</resourceadapter-version>

                           <license>
                              <description>
                        Copyright 2009 Red Hat, Inc.
                         Red Hat licenses this file to you under the Apache License,
version
                         2.0 (the "License"); you may not use this file except in
compliance
```

```
                           with the License.  You may obtain a copy of the License at
                             http://www.apache.org/licenses/LICENSE-2.0
                           Unless required by applicable law or agreed to in writing,
 software
                           distributed under the License is distributed on an "AS IS" BASIS,
                           WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or
                           implied.  See the License for the specific language governing
                           permissions and limitations under the License.
                                </description>
                                <license-required>true</license-required>
                             </license>

                             <resourceadapter>
                                <resourceadapter-
class>org.hornetq.ra.HornetQResourceAdapter</resourceadapter-class>
                                <config-property>
                                   <description>The transport type</description>
                                   <config-property-name>ConnectorClassName</config-
property-name>
                                   <config-property-type>java.lang.String</config-property-
type>
                                   <config-property-
value>org.hornetq.core.remoting.impl.netty.NettyConnectorFactory</config-property-value>
                                </config-property>
                                <config-property>
                                   <description>The transport configuration. These values
 must be in the form of key=val;key=val;</description>
                                   <config-property-name>ConnectionParameters</config-
property-name>
                                   <config-property-type>java.lang.String</config-property-
type>
                                    <config-property-value>host=127.0.0.1;port=5445</config-
property-value>
                                </config-property>

                                <outbound-resourceadapter>
                                   <connection-definition>
                                      <managedconnectionfactory-
class>org.hornetq.ra.HornetQRAManagedConnectionFactory</managedconnectionfactory-class>

                                      <config-property>
                                         <description>The default session type</description>
                                         <config-property-name>SessionDefaultType</config-
property-name>
                                         <config-property-type>java.lang.String</config-
property-type>
                                         <config-property-value>javax.jms.Queue</config-
property-value>
                                      </config-property>
                                      <config-property>
                                         <description>Try to obtain a lock within specified
 number of seconds; less than or equal to 0 disable this functionality</description>
                                         <config-property-name>UseTryLock</config-property-
name>
                                         <config-property-type>java.lang.Integer</config-
property-type>
                                         <config-property-value>0</config-property-value>
                                      </config-property>

                                      <connectionfactory-
interface>org.hornetq.ra.HornetQRAConnectionFactory</connectionfactory-interface>
                                      <connectionfactory-impl-
class>org.hornetq.ra.HornetQRAConnectionFactoryImpl</connectionfactory-impl-class>
                                      <connection-interface>javax.jms.Session</connection-
interface>
                                      <connection-impl-
class>org.hornetq.ra.HornetQRASession</connection-impl-class>
```

```
                                    </connection-definition>
                                    <transaction-support>XATransaction</transaction-support>
                                    <authentication-mechanism>
                                       <authentication-mechanism-type>BasicPassword</
authentication-mechanism-type>
                                       <credential-
interface>javax.resource.spi.security.PasswordCredential</credential-interface>
                                    </authentication-mechanism>
                                    <reauthentication-support>false</reauthentication-
support>
                                 </outbound-resourceadapter>

                                 <inbound-resourceadapter>
                                    <messageadapter>
                                       <messagelistener>
                                          <messagelistener-type>javax.jms.MessageListener</
messagelistener-type>
                                          <activationspec>
                                             <activationspec-
class>org.hornetq.ra.inflow.HornetQActivationSpec</activationspec-class>
                                             <required-config-property>
                                                 <config-property-name>destination</config-
property-name>
                                             </required-config-property>
                                          </activationspec>
                                       </messagelistener>
                                    </messageadapter>
                                 </inbound-resourceadapter>

                              </resourceadapter>
                           </connector>
```

The important part of this configuration is the part in bold, i.e. <config-property-value>host=127.0.0.1;port=5445</config-property-value>. This should be configured to the host and port of the remote HornetQ server.

At this point you should be able to now deploy MDBs that consume from the remote server. You will however, have to make sure that your MDBs have the annotation **`@ResourceAdapter("hornetq-ra.rar")`** added, this is illustrated in the *Section 32.1, "Configuring Message-Driven Beans"* section. If you don't want to add this annotation then you can delete the generic resource adapter **`jms-ra.rar`** and rename the **`hornetq-ra.rar`** to this.

If you also want to use the remote HornetQ server for outgoing connections, i.e. sending messages, then do the following:

- Create a file called **`hornetq-ds.xml`** in the **`deploy`** directory (in fact you can call this anything you want as long as it ends in **`-ds.xml`**). Then add the following:

```
<connection-factories>
<!--JMS XA Resource adapter, use this for outbound JMS connections. Inbound connections
 are defined at the @MDB activation or at the resource-adapter properties. -->
   <tx-connection-factory>
      <jndi-name>RemoteJmsXA</jndi-name>
      <xa-transaction/>
      <rar-name>hornetq-ra.rar</rar-name>
      <connection-definition>org.hornetq.ra.HornetQRAConnectionFactory</connection-
definition>
      <config-property name="SessionDefaultType" type="java.lang.String">javax.jms.Topic</
config-property>
      <config-property name="ConnectorClassName"
 type="java.lang.String">org.hornetq.core.remoting.impl.netty.NettyConnectorFactory</
config-property>
```

```
      <config-property name="ConnectionParameters"
 type="java.lang.String">host=127.0.0.1;port=5445</config-property>
      <max-pool-size>20</max-pool-size>
   </tx-connection-factory>
</connection-factories>
```

Again you will see that the host and port are configured here to match the remote HornetQ servers configuration. The other important attributes are:

- jndi-name - This is the name used to look up the JMS connection factory from within your Java EE client
- rar-name - This should match the directory that you created to hold the Resource Adapter configuration

Now you should be able to send messages using the JCA JMS connection pooling within an XA transaction.

32.5.2. Configuring JBoss 5 MDB

The steps to do this are exactly the same as for JBoss 4, you will have to create a jboss.xml definition file for your MDB with the following entry:

```
<message-driven>
   <ejb-name>MyMDB</ejb-name>
   <resource-adapter-name>jms-ra.rar</resource-adapter-name>
</message-driven>
```

Also you will need to edit the **`standardjboss.xml`** file and uncomment the section with the following 'Uncomment to use JMS message inflow from **`jms-ra.rar`**' and then comment out the invoker-proxy-binding called 'message-driven-bean'.

32.6. High Availability JNDI (HA-JNDI)

If you are using JNDI to look-up JMS queues, topics and connection factories from a cluster of servers, it is likely you will want to use HA-JNDI so that your JNDI look-ups will continue to work if one or more of the servers in the cluster fail.

HA-JNDI is a JBoss Application Server service which allows you to use JNDI from clients without them having to know the exact JNDI connection details of every server in the cluster. This service is only available if using a cluster of JBoss Application Server instances.

To use it use the following properties when connecting to JNDI.

```
Hashtable<String, String>
  jndiParameters = new Hashtable<String, String>();
jndiParameters.put("java.naming.factory.initial",
    "org.jnp.interfaces.NamingContextFactory");
jndiParameters.put("java.naming.factory.url.pkgs=",
    "org.jboss.naming:org.jnp.interfaces");

initialContext = new InitialContext(jndiParameters);
```

For more information on using HA-JNDI see the Clustering section of the *Administration and Configuration Guide* for this release of JBoss Enterprise Application Platform.

32.7. XA Recovery

XA recovery deals with system or application failures to ensure that of a transaction are applied consistently to all resources affected by the transaction, even if any of the application processes or the machine hosting them crash or lose network connectivity. For more information on XA Recovery,please refer to the JBoss Transactions documentation provided with this release of JBoss Enterprise Application Platform.

When HornetQ is integrated with JBoss AS, it can take advantage of JBoss Transactions to provide recovery of messaging resources. If messages are involved in a XA transaction, in the event of a server crash, the recovery manager will ensure that the transactions are recovered and the messages will either be committed or rolled back (depending on the transaction outcome) when the server is restarted.

32.7.1. XA Recovery Configuration

To enable HornetQ's XA Recovery, the Recovery Manager must be configured to connect to HornetQ to recover its resources. The following property must be added to the **`jta`** section of **`conf/jbossts-properties.xml`** of JBoss AS profiles:

```
<properties depends="arjuna" name="jta">
   ...

   <property name="com.arjuna.ats.jta.recovery.XAResourceRecovery.HornetQ1"
      value="org.hornetq.jms.server.recovery.HornetQXAResourceRecovery;[connection
 configuration]"/>
</properties>
```

The **`[connection configuration]`** contains all the information required to connect to HornetQ node under the form **`[connector factory class name],[user name], [password], [connector parameters]`**.

- **`[connector factory class name]`** corresponds to the name of the **`ConnectorFactory`** used to connect to HornetQ. Values can be **`org.hornetq.core.remoting.impl.invm.InVMConnectorFactory`** or **`org.hornetq.core.remoting.impl.netty.NettyConnectorFactory`**
- **`[user name]`** is the user name to create a client session. It is optional
- **`[password]`** is the password to create a client session. It is mandatory only if the user name is specified
- **`[connector parameters]`** is a list of comma-separated key=value pair which are passed to the connector factory (see *Chapter 16, Configuring the Transport* for a list of the transport parameters).

Note

HornetQ must have a valid acceptor which corresponds to the connector specified in **`conf/jbossts-properties.xml`**.

32.7.1.1. Configuration Settings

If HornetQ is configured with a default in-vm acceptor:

```
<acceptor name="in-vm">
    <factory-class>org.hornetq.core.remoting.impl.invm.InVMAcceptorFactory</factory-class>
</acceptor>
```

the corresponding configuration in **`conf/jbossts-properties.xml`** is:

```
<property name="com.arjuna.ats.jta.recovery.XAResourceRecovery.HORNETQ1"
 value="org.hornetq.jms.server.recovery.HornetQXAResourceRecovery;
        org.hornetq.core.remoting.impl.invm.InVMConnectorFactory"/>
```

If it is now configured with a netty acceptor on a non-default port:

```
<acceptor name="netty">
  <factory-class>
    org.hornetq.core.remoting.impl.netty.NettyAcceptorFactory
  </factory-class>
  <param key="port" value="8888"/>
</acceptor>
```

the corresponding configuration in **`conf/jbossts-properties.xml`** is:

```
<property name="com.arjuna.ats.jta.recovery.XAResourceRecovery.HORNETQ1"
  value="org.hornetq.jms.server.recovery.HornetQXAResourceRecovery;
        org.hornetq.core.remoting.impl.netty.NettyConnectorFactory, , , port=8888"/>
```

Note

Note the additional commas to skip the user and password before connector parameters

If the recovery must use **`admin, adminpass`**, the configuration would have been:

```
                    <property name="com.arjuna.ats.jta.recovery.XAResourceRecovery.HORNETQ1"

  value="org.hornetq.jms.server.recovery.HornetQXAResourceRecovery;
        org.hornetq.core.remoting.impl.netty.NettyConnectorFactory, admin, adminpass,
 port=8888"/>
```

Configuring HornetQ with an InVM acceptor and configuring the Recovery Manager with an InVM connector is the recommended way to enable XA Recovery.

32.7.2. Example

See *Section 11.3.9, "XA Recovery"* which shows how to configure XA Recovery and recover messages after a server crash.

The JMS Bridge

HornetQ includes a fully-functional JMS message bridge, which consumes messages from a source queue or topic and sends them to a target queue or topic, usually on a different server.

The source and target servers do not have to be in the same cluster which makes bridging suitable for reliably sending messages from one cluster to another, for instance across a WAN, and where the connection may be unreliable.

The bridge can also be used to bridge messages from other non-HornetQ JMS servers that are JMS 1.1 compliant.

A JMS bridge is not a Core bridge

A JMS bridge can be used to bridge any two JMS 1.1 compliant JMS providers and uses the JMS API. A Core bridge (described in *Chapter 36, Core Bridges*) is used to bridge any two HornetQ instances and uses the core API. A core bridge will typically provide better performance than a JMS bridge, and provides once-and-only-once delivery guarantees without using XA.

The bridge has built-in resilience to failure so if the source or target server connection is lost (for example, due to network failure), the bridge will attempt to reconnect to the target server until it comes back online, at which point operations will resume as normal.

The bridge can be configured with an optional JMS selector, so it will only consume messages matching that JMS selector

It can be configured to consume from a queue or a topic. When it consumes from a topic it can be configured to consume using a non-durable or durable subscription.

Typically, the bridge is deployed by JBoss Microcontainer via a beans configuration file, **`jms-bridge-jboss-beans.xml`**. This is then deployed inside the application server. The following is a beans file that bridges two destinations on the same server.

```
<?xml version="1.0" encoding="UTF-8"?>

<deployment xmlns="urn:jboss:bean-deployer:2.0">

  <bean name="JMSBridge" class="org.hornetq.api.jms.bridge.impl.JMSBridgeImpl">
     <!-- HornetQ must be started before the bridge -->
     <depends>HornetQServer</depends>
     <constructor>
         <!-- Source ConnectionFactory Factory -->
         <parameter>
             <inject bean="SourceCFF"/>
         </parameter>
         <!-- Target ConnectionFactory Factory -->
         <parameter>
             <inject bean="TargetCFF"/>
         </parameter>
         <!-- Source DestinationFactory -->
         <parameter>
             <inject bean="SourceDestinationFactory"/>
         </parameter>
         <!-- Target DestinationFactory -->
         <parameter>
             <inject bean="TargetDestinationFactory"/>
         </parameter>
         <!-- Source User Name (no username here) -->
         <parameter><null /></parameter>
```

```
        <!-- Source Password (no password here)-->
        <parameter><null /></parameter>
        <!-- Target User Name (no username here)-->
        <parameter><null /></parameter>
        <!-- Target Password (no password here)-->
        <parameter><null /></parameter>
        <!-- Selector -->
        <parameter><null /></parameter>
        <!-- Failure Retry Interval (in ms) -->
        <parameter>5000</parameter>
        <!-- Max Retries -->
        <parameter>10</parameter>
        <!-- Quality Of Service -->
        <parameter>ONCE_AND_ONLY_ONCE</parameter>
        <!-- Max Batch Size -->
        <parameter>1</parameter>
        <!-- Max Batch Time (-1 means infinite) -->
        <parameter>-1</parameter>
        <!-- Subscription name (no subscription name here)-->
        <parameter><null /></parameter>
        <!-- Client ID  (no client ID here)-->
        <parameter><null /></parameter>
        <!-- Add MessageID In Header -->
        <parameter>true</parameter>
        <!-- register the JMS Bridge in the AS MBeanServer -->
        <parameter>
            <inject bean="MBeanServer"/>
        </parameter>
        <parameter>org.hornetq:service=JMSBridge</parameter>
      </constructor>
    <property name="transactionManager">
        <inject bean="RealTransactionManager"/>
    </property>
</bean>

<!-- SourceCFF describes the ConnectionFactory used to connect to the
     source destination -->
<bean name="SourceCFF"
    class="org.hornetq.api.jms.bridge.impl.JNDIConnectionFactoryFactory">
    <constructor>
        <parameter>
            <inject bean="JNDI" />
        </parameter>
        <parameter>/ConnectionFactory</parameter>
    </constructor>
</bean>

<!-- TargetCFF describes the ConnectionFactory used to connect to the
target destination -->
<bean name="TargetCFF"
    class="org.hornetq.api.jms.bridge.impl.JNDIConnectionFactoryFactory">
    <constructor>
        <parameter>
            <inject bean="JNDI" />
        </parameter>
        <parameter>/ConnectionFactory</parameter>
    </constructor>
</bean>

<!-- SourceDestinationFactory describes the Destination used as the source -->
<bean name="SourceDestinationFactory"
    class="org.hornetq.api.jms.bridge.impl.JNDIDestinationFactory">
    <constructor>
        <parameter>
            <inject bean="JNDI" />
        </parameter>
        <parameter>/queue/source</parameter>
```

```
      </constructor>
   </bean>

   <!-- TargetDestinationFactory describes the Destination used as the target -->
   <bean name="TargetDestinationFactory"
       class="org.hornetq.api.jms.bridge.impl.JNDIDestinationFactory">
      <constructor>
          <parameter>
              <inject bean="JNDI" />
          </parameter>
          <parameter>/queue/target</parameter>
      </constructor>
   </bean>

   <!-- JNDI is a Hashtable containing the JNDI properties required -->
   <!-- to connect to the sources and targets JMS resources         -->
   <bean name="JNDI" class="java.util.Hashtable">
    <constructor class="java.util.Map">
       <map class="java.util.Hashtable" keyClass="String"
                                        valueClass="String">
          <entry>
             <key>java.naming.factory.initial</key>
             <value>org.jnp.interfaces.NamingContextFactory</value>
          </entry>
          <entry>
             <key>java.naming.provider.url</key>
             <value>jnp://localhost:1099</value>
          </entry>
          <entry>
             <key>java.naming.factory.url.pkgs</key>
             <value>org.jboss.naming:org.jnp.interfaces"</value>
          </entry>
       </map>
    </constructor>
   </bean>

   <bean name="MBeanServer" class="javax.management.MBeanServer">
    <constructor factoryClass="org.jboss.mx.util.MBeanServerLocator"
                 factoryMethod="locateJBoss"/>
   </bean>
</deployment>
```

33.1. JMS Bridge Parameters

The **JMSBridge** bean, as shown in the example above, is configured via parameters passed to its constructor in a particular order. This order, and a description of each parameter, is outlined in the list following.

Note

To leave a parameter unspecified (for example, if the authentication is anonymous or no message selector is provided), use **<null />** for the unspecified parameter value.

`Source Connection Factory Factory`
: Injects the **SourceCFF** bean defined in the **jms-bridge-jboss-beans.xml** file, which creates the **ConnectionFactory**.

`Target Connection Factory Factory`
: Injects the **TargetCFF** bean defined in the **jms-bridge-jboss-beans.xml** file, which creates the target **ConnectionFactory**.

`Source Destination Factory Factory`
Injects the **`SourceDestinationFactory`** bean defined in the **`jms-bridge-jboss-beans.xml`** file, which creates the source **`Destination`**.

`Target Destination Factory Factory`
Injects the **`TargetDestinationFactory`** bean defined in the **`jms-bridge-jboss-beans.xml`** file, which creates the target **`Destination`**.

`Source User Name`
Defines the username used to create the source connection.

`Source Password`
Defines the password for the username used to create the source connection.

`Target User Name`
Defines the username used to create the target connection.

`Target Password`
Defines the password of the username used to create the target connection.

`Selector`
Specifies a JMS selector expression used when consuming messages from the source destination. Only messages that match the selector expression will be bridged from the source to the target destination. The selector expression must follow the *JMS selector syntax*[1].

`Failure Retry Interval`
Specifies the time in milliseconds to wait in between attempting to recreate connections to the source or target servers when the bridge detects a connection failure.

`Max Retries`
Specifies the number of times to attempt to recreate connections to the source or target servers when the bridge detects a connection failure. The bridge will stop trying to reconnect after this number of tries. **`-1`** means 'try forever'.

`Quality of Service`
Defines the quality of service mode. The possible values are:

- **`AT_MOST_ONCE`**
- **`DUPLICATES_OK`**
- **`ONCE_AND_ONLY_ONCE`**

See *Section 33.4, "Quality Of Service Modes"* for a explanation of these modes.

`Max Batch Size`
Defines the maximum number of messages that should be consumed from the source connection before the messages are sent in a batch to the target destination. Its value must be **`1`** or greater.

`Max Batch Time`
Defines the number of milliseconds to wait before sending a batch to the target destination, even if the number of messages consumed has not reached `MaxBatchSize`. Its value must be **`1`** or greater, or **`-1`** to specify 'wait forever'.

[1] http://java.sun.com/j2ee/1.4/docs/api/javax/jms/Message.html

`Subscription Name`
: If the source destination is a topic, and you want to consume from the topic with a durable subscription, this parameter defines the durable subscription name.

`Client ID`
: If the source destination is a topic, and you want to consume from the topic with a durable subscription, this parameter defines the JMS client ID to use when creating or looking up the durable subscription.

`Add MessageID In Header`
: When **`true`**, the original message's message ID is appended to the message sent to the destination in the **`HORNETQ_BRIDGE_MSG_ID_LIST`** header. If the message is bridged multiple times, each message ID is appended. This lets you use a distributed response pattern.

Note

When a message is received, a response can be sent using the correlation ID of the first message ID so that when the original sender receives the response it is able to correlate the message.

`MBean Server`
: Set this to the place where the JMS Bridge is registered (the application server MBeanServer) to manage the JMS Bridge with JMX.

`ObjectName`
: If `MBeanServer` is set, this parameter must be set to define the name used to register the JMS Bridge MBean. This name must be unique.

33.2. Source and Target Connection Factories

The source and target connection factory factories are used to create the connection factory that creates the connection for the source or target server.

The example configuration code uses the default implementation provided by HornetQ, which looks up the connection factory using JNDI. For other JMS providers, a different implementation may be required. To do so, implement the **`org.hornetq.jms.bridge.ConnectionFactoryFactory`** interface.

33.3. Source and Target Destination Factories

These create or look up the destinations.

The example configuration code uses the default implementation provided by HornetQ that looks up the destination using JNDI.

Different implementations can be provided with the **`org.hornetq.jms.bridge.DestinationFactory`** interface.

33.4. Quality Of Service Modes

The quality of service modes used by the bridge are described here in more detail.

33.4.1. AT_MOST_ONCE

Messages reach the destination once at most. Messages are consumed from the source and acknowledged before they are sent to the destination. If failure occurs between leaving the source and arriving at the destination, messages can be lost. This mode is available for both durable and non-durable messages.

33.4.2. DUPLICATES_OK

Messages are consumed from the source and acknowledged after they are successfully sent to the destination. If failure occurs after messages are sent, but before acknowledgment is returned, messages can be sent again once the system recovers, so the destination may receive duplicates after a failure. This mode is available for both durable and non-durable messages.

33.4.3. ONCE_AND_ONLY_ONCE

Messages reach the destination exactly once. If both source and destination are on the same HornetQ server instance, this mode sends and acknowledges messages as part of the same local transaction. If the source and destination are on different servers, this mode enlists the sending and consumption sessions in a JTA transaction.

This JTA transaction is controlled by JBoss Transactions JTA, a full-recovery transaction manager which provides a very high degree of durability. If JTA is required, both supplied connection factories must be **`XAConnectionFactory`** implementations.

This is likely to be the slowest mode, since it requires extra persistence. This mode is only available for durable messages.

Note

For some applications, once and only once semantics could be provided by setting the **`DUPLICATES_OK`** mode and then checking for and discarding duplicate messages on the destination. This approach is not as reliable as using **`ONCE_AND_ONLY_ONCE`** mode, but may be a useful alternative.

33.4.4. Time outs and the JMS bridge

There is a possibility that the target or source server will not be available at some point in time. If this occurs then the bridge will try **`Max Retries`** to reconnect every **`Failure Retry Interval`** milliseconds as specified in the JMS Bridge definition.

However since a third party JNDI is used, in this case the JBoss naming server, it is possible for the JNDI lookup to hang if the network were to disappear during the JNDI lookup. To stop this occurring the JNDI definition can be configured to time out if this occurs. To do this set the **`jnp.timeout`** and the **`jnp.sotimeout`** on the Initial Context definition. The first sets the connection timeout for the initial connection and the second the read timeout for the socket.

Note

Once the initial JNDI connection has succeeded all calls are made using RMI. If you want to control the timeouts for the RMI connections then this can be done via system properties. JBoss uses Sun's RMI and the properties can be found *here*[2]. The default connection timeout is 10 seconds and the default read timeout is 18 seconds.

If you implement your own factories for looking up JMS resources then you will have to bear in mind timeout issues.

33.4.5. Examples

Please see *Section 11.3.5, "JMS Bridge"* which shows how to configure and use a JMS Bridge with JBoss AS to send messages to the source destination and consume them from the target destination.

[2] http://java.sun.com/j2se/1.5.0/docs/guide/rmi/sunrmiproperties.html

Client Reconnection and Session Reattachment

HornetQ clients can be configured to automatically reconnect or re-attach to the server if a failure is detected in the server-client connection.

34.1. 100% Transparent session re-attachment

If failure is transient (that is, due to a temporary network failure) and the target server was not restarted, sessions will still exist on the server as long as the `connection-ttl` has not expired (see *Chapter 17, Detecting Dead Connections* for details).

In this case, HornetQ automatically re-attaches the client sessions to the server sessions when the connection reconnects. This is done 100% transparently and the client can continue as before.

As HornetQ clients send commands to their servers, they store each sent command in an in-memory buffer. When connection failure occurs and the client re-attaches to its server, the server passes the ID of the last command it successfully received back to the client as part of the re-attachment protocol. If the client had attempted to send any other commands, it can resend those commands from its buffer so that client and server can reconcile their differences.

The `ConfirmationWindowSize` parameter defines the size of the buffer in bytes. When the server has received `ConfirmationWindowSize` bytes of commands and processed them it will send back a command confirmation to the client. The client can then remove confirmed commands from the buffer.

If you are using JMS, and the JMS service on the server is being used to load your JMS connection factory instances into JNDI, then the `ConfirmationWindowSize` parameter can be set in the **`hornetq-jms.xml`** file. If you are using JMS, but not JNDI, you can set these values directly on the **`HornetQConnectionFactory`** instance with the appropriate setter method. If you are using Core, you can set these values directly on the **`ClientSessionFactory`** instance with the appropriate setter method.

Setting `ConfirmationWindowSize` to **`-1`** (the default) disables buffering and prevents re-attachment from occurring, forcing reconnect instead.

34.2. Session reconnection

If the server is restarted after crashing or being stopped, sessions will no longer exist on the server and it will not be possible to re-attach completely transparently.

In this case, HornetQ automatically reconnects the connection and recreates any sessions and consumers on the server based on the sessions and consumers of the client. (This process is identical to failover onto a backup server.) Client reconnection is also used internally by components such as core bridges to let them reconnect to their target servers. See *Section 39.2.1, “Automatic Client Failover”* for a full explanation of how transacted and non-transacted sessions are reconnected during failover, and the requirements of once-and-only-once delivery guarantees.

34.3. Configuring reconnection/reattachment attributes

Client reconnection is configured using the following parameters:

`retry-interval`
: An optional parameter that defines the period in milliseconds between subsequent reconnection attempts, if connection to the target server has failed. The default value is **`2000`** milliseconds.

`retry-interval-multiplier`
: An optional parameter that defines a multiplier to apply to the time since the last retry in order compute the time to the next retry. This means that the time between retries increases exponentially with each unsuccessful attempt. The default value is **`1.0`**, which spaces each reconnect attempt at equal intervals.

`max-retry-interval`
: An optional parameter that defines the maximum interval in milliseconds between retries. Even when `retry-interval-multiplier` is set, the interval between retries will not be greater than this value. The default value is **`2000`** milliseconds.

`reconnect-attempts`
: An optional parameter that defines the total number of reconnect attempts to make before ceasing to attempt reconnection and shutting down. A value of **`-1`** sets an unlimited number of attempts. The default value is **`0`**.

If you are using JMS and the JMS Service on the server is being used to load your JMS connection factory instances into JNDI, you can specify these parameters in **`hornetq-jms.xml`** like so:

```
<connection-factory name="ConnectionFactory">
   <connectors>
      <connector-ref connector-name="netty"/>
   </connectors>
   <entries>
      <entry name="ConnectionFactory"/>
      <entry name="XAConnectionFactory"/>
   </entries>
   <retry-interval>1000</retry-interval>
   <retry-interval-multiplier>1.5</retry-interval-multiplier>
   <max-retry-interval>60000</max-retry-interval>
   <reconnect-attempts>1000</reconnect-attempts>
</connection-factory>
```

If you are using JMS, but instantiating your JMS connection factory directly, you can specify the parameters using the appropriate setter methods on the **`HornetQConnectionFactory`** immediately after its creation.

If you are using the core API and instantiating the **`ClientSessionFactory`** instance directly, you can also specify the parameters using the appropriate setter methods on the **`ClientSessionFactory`** immediately after its creation.

If your client does manage to reconnect but the session is no longer available on the server, for instance if the server has been restarted or it has timed out, then the client will be unable to re-attach, and any **`ExceptionListener`** or **`FailureListener`** instances registered on the connection or session will be called.

ExceptionListeners and SessionFailureListeners

Registered JMS **`ExceptionListener`** or Core **`SessionFailureListener`** instances will be called when a client reconnects or re-attaches.

Diverting and Splitting Message Flows

Diverts are objects that transparently divert messages routed to one address to some other address, without making any changes to any client application logic.

Diverts can be *exclusive* (messages are diverted to the new address and to not go to the previous address at all), or *non-exclusive* (a copy of the message is sent to the new address, and the original message goes to the old address). Non-exclusive diverts can therefore be used for *splitting* message flows, for example, when every order sent to an order queue must be monitored.

Diverts can also be configured with a filter, so that only messages that match the filter are diverted.

Diverts can also be configured to apply a **`Transformer`**. If specified, all diverted messages can be transformed by this **`Transformer`**.

Diverts only move messages to addresses on the same server, but when used in combination with bridges more complex routings can be set up. One common use case to "divert" to a different server is to divert messages to a local store-and-forward queue and then set up a bridge to consume from that queue and forward consumed messages to an address on another server. The diverts on a server can be thought of as a routing table for messages. Combining diverts with bridges allow you to create a distributed network of reliable routing connections between multiple geographically distributed servers.

Diverts are defined in the **`hornetq-configuration.xml`** file. There can be zero or more diverts in the file.

See *Section 11.1.13, “Divert”* for an example of using and configuring diverts in HornetQ.

35.1. Exclusive Diverts

An exclusive divert diverts all matching messages that are routed to the old address to the new address. Matching messages do not get routed to the old address.

Diverts are defined in **`hornetq-configuration.xml`**, like so:

```
<divert name="prices-divert">
   <address>jms.topic.priceUpdates</address>
   <forwarding-address>jms.queue.priceForwarding</forwarding-address>
   <filter string="office='New York'"/>
   <transformer-class-name>
    org.hornetq.jms.example.AddForwardingTimeTransformer
   </transformer-class-name>
   <exclusive>true</exclusive>
</divert>
```

The **`prices-divert`** divert specified above diverts any messages sent to the address **`jms.topic.priceUpdates`** (a local JMS Topic called **`priceUpdates`**) to another local address **`jms.queue.priceForwarding`** (a local JMS Queue called **`priceForwarding`**).

A `filter string` is also specified so that only messages with the message property **`office='New York'`** are diverted. All other messages continue to be routed to their usual address. The filter string is optional; if it is not specified, all messages are diverted.

The transformer class is also optional. When specified, transformation is executed for each matching message. This allows you to change the message body or properties before the message is diverted. The example above uses a transformation to add a header recording the time the divert occurred.

As a whole, the above example diverts messages to a local store-and-forward queue, which is configured with a bridge, which forwards the message to an address on another HornetQ server. See the HornetQ examples for further details.

35.2. Non-exclusive Diverts

Non-exclusive diverts forward a copy of a message to a new address, allowing the original message to continue to the previous address. They can be thought of as splitting message flow.

Non-exclusive diverts are configured similarly to exclusive diverts, with an optional filter and transformer, like so:

```
<divert name="order-divert">
   <address>jms.queue.orders</address>
   <forwarding-address>jms.topic.spyTopic</forwarding-address>
   <exclusive>false</exclusive>
</divert>
```

The **`order-divert`** example copies every message sent to the **`jms.queue.orders`** address (a JMS Queue called **`orders`**) and forwards the copy to a local address **`jms.topic.SpyTopic`** (a JMS Topic called **`SpyTopic`**).

Core Bridges

The function of a bridge is to consume messages from a source queue, and forward them to a target address, typically on a different HornetQ server.

The source and target servers do not have to be in the same cluster which makes bridging suitable for reliably sending messages from one cluster to another, for instance across a WAN, or Internet and where the connection may be unreliable.

The bridge has built in resilience to failure so if the target server connection is lost, e.g. due to network failure, the bridge will retry connecting to the target until it comes back online. When it comes back online it will resume operation as normal.

In summary, bridges are a way to reliably connect two separate HornetQ servers together. With a core bridge both source and target servers must be HornetQ servers.

Bridges can be configured to provide *once and only once* delivery guarantees even in the event of the failure of the source or the target server. They do this by using duplicate detection (described in *Chapter 37, Duplicate Message Detection*).

Note

Although they have similar function, don't confuse core bridges with JMS bridges!

Core bridges are for linking a HornetQ node with another HornetQ node and do not use the JMS API. A JMS Bridge is used for linking any two JMS 1.1 compliant JMS providers. So, a JMS Bridge could be used for bridging to or from different JMS compliant messaging system. It's always preferable to use a core bridge where possible. Core bridges use duplicate detection to provide *once and only once* guarantees. To provide the same guarantee using a JMS bridge, an XA which has a higher overhead and is more complex to configure would need to be used.

36.1. Configuring Bridges

Bridges are configured in **`hornetq-configuration.xml`**. An example follows (this is actually from the bridge example):

```
<bridge name="my-bridge">
   <queue-name>jms.queue.sausage-factory</queue-name>
   <forwarding-address>jms.queue.mincing-machine</forwarding-address>
   <filter-string="name='aardvark'"/>
   <transformer-class-name>
        org.hornetq.jms.example.HatColourChangeTransformer
   </transformer-class-name>
   <retry-interval>1000</retry-interval>
   <retry-interval-multiplier>1.0</retry-interval-multiplier>
   <reconnect-attempts>-1</reconnect-attempts>
   <failover-on-server-shutdown>false</failover-on-server-shutdown>
   <use-duplicate-detection>true</use-duplicate-detection>
   <confirmation-window-size>10000000</confirmation-window-size>
   <connector-ref connector-name="remote-connector"
        backup-connector-name="backup-remote-connector"/>
   <user>foouser</user>
   <password>foopassword</password>
</bridge>
```

In the above example we have shown all the parameters its possible to configure for a bridge. In practice, many of the defaults may be used, therefore it will not be necessary to specify them all explicitly.

The parameters are described in the following list.

Core Bridge Parameters

`name attribute`
All bridges must have a unique name in the server.

`queue-name`
This is the unique name of the local queue that the bridge consumes from, it is a mandatory parameter.

The queue must already exist by the time the bridge is instantiated at start-up.

Note

If using JMS then normally the JMS configuration **`hornetq-jms.xml`** is loaded after the core configuration file **`hornetq-configuration.xml`** is loaded. If the bridge is consuming from a JMS queue then ensure that the JMS queue is also deployed as a core queue in the core configuration. Refer to the bridge example for an example of this.

`forwarding-address`
This is the address on the target server that the message will be forwarded to. If a forwarding address is not specified, then the original address of the message will be retained.

`filter-string`
An optional filter string can be supplied. If specified, only messages which match the filter expression specified in the filter string will be forwarded. The filter string follows the HornetQ filter expression syntax described in *Chapter 14, Filter Expressions*.

`transformer-class-name`
An optional transformer-class-name can be specified. This is the name of a user-defined class which implements the `org.hornetq.core.server.cluster.Transformer` interface.

If this is specified then the transformer's `transform()` method will be invoked with the message before it is forwarded. This allows the opportunity to transform the message header or body before forwarding

`retry-interval`
This optional parameter determines the period in milliseconds between subsequent reconnection attempts, if the connection to the target server has failed. The default value is **`2000`**milliseconds.

`retry-interval-multiplier`
This optional parameter determines a multiplier to apply to the time since the last retry in order to compute the time to the next retry.

This allows an *exponential backoff* between retry attempts to be implemented.

For example:

If `retry-interval` is set to **`1000`** ms and `retry-interval-multiplier` is set to **`2.0`**, then, if the first reconnect attempt fails, there will be a wait of **`1000`** ms, then **`2000`** ms and then **`4000`** ms between subsequent reconnection attempts.

The default value is **1.0** meaning each reconnect attempt is spaced at equal intervals.

`reconnect-attempts`
This optional parameter determines the total number of reconnect attempts the bridge will make before giving up and shutting down. A value of **-1** signifies an unlimited number of attempts. The default value is **-1**.

`failover-on-server-shutdown`
This optional parameter determines whether the bridge will attempt to failover onto a backup server (if specified) when the target server is cleanly shutdown rather than crashed.

The bridge connector can specify both a live and a backup server. If it specifies a backup server and this parameter is set to **true**, then if the target server is *cleanly* shutdown the bridge connection will attempt to failover onto its backup. If the bridge connector has no backup server configured, this parameter has no effect.

This parameter is useful when occasionally a bridge configured with a live and a backup target server is required, but failover to the backup is not required if the live server is simply taken down temporarily for maintenance.

The default value for this parameter is **false**.

`use-duplicate-detection`
This optional parameter determines whether the bridge will automatically insert a duplicate id property into each message that it forwards.

Doing so, allows the target server to perform duplicate detection on messages it receives from the source server. If the connection fails or the server crashes, when the bridge resumes it will resend unacknowledged messages. This might result in duplicate messages being sent to the target server. Enabling duplicate detection allows these duplicates to be screened out and ignored.

This allows the bridge to provide a *once and only once* delivery guarantee without using heavyweight methods such as XA (Refer to *Chapter 37, Duplicate Message Detection* for more information).

The default value for this parameter is **true**.

`confirmation-window-size`
This optional parameter determines the `confirmation-window-size` to use for the connection used to forward messages to the target node. This attribute is described in section *Chapter 34, Client Reconnection and Session Reattachment*.

Warning

When using the bridge to forward messages from a queue which has a max-size-bytes set it is important that confirmation-window-size is less than or equal to `max-size-bytes` to prevent the flow of messages from ceasing.

`connector-ref`
his mandatory parameter determines which *connector* pair the bridge will use to actually make the connection to the target server.

A *connector* encapsulates knowledge of what transport to use (TCP, SSL, HTTP etc) as well as the server connection parameters (host, port etc). For more information on connectors and their configuration, refer to *Chapter 16, Configuring the Transport*.

The `connector-ref` element can be configured with two attributes:

- `connector-name`. This references the name of a connector defined in the core configuration file **`hornetq-configuration.xml`**. The bridge will use this connector to make its connection to the target server. This attribute is mandatory.
- `backup-connector-name`. This optional parameter also references the name of a connector defined in the core configuration file **`hornetq-configuration.xml`**. It represents the connector that the bridge will fail-over to if it detects that the live server connection has failed. If this is specified and `failover-on-server-shutdown` is set to **`true`** then it will also attempt failover onto this connector if the live target server is cleanly shut-down.

`user`
: This optional parameter determines the user name to use when creating the bridge connection to the remote server. If it is not specified the default cluster user specified by `cluster-user` in **`hornetq-configuration.xml`** will be used.

`password`
: This optional parameter determines the password to use when creating the bridge connection to the remote server. If it is not specified, the default cluster password specified by `cluster-password` in **`hornetq-configuration.xml`** will be used.

Duplicate Message Detection

HornetQ includes powerful automatic duplicate message detection, filtering out duplicate messages without having to code duplicate detection logic at the application level. This chapter will explain what duplicate detection is, how HornetQ uses it and how and where to configure it.

When sending messages from a client to a server, or indeed from a server to another server, if the target server or connection fails sometime after sending the message, but before the sender receives a response that the send (or commit) was processed successfully then the sender cannot know for sure if the message was sent successfully to the address.

If the target server or connection failed after the send was received and processed but before the response was sent back then the message will have been sent to the address successfully, but if the target server or connection failed before the send was received and finished processing then it will not have been sent to the address successfully. From the senders point of view, it is not possible to distinguish between these two cases.

When the server recovers, this leaves the client in a difficult situation. It knows the target server failed, but it does not know if the last message reached its destination satisfactorily. If it decides to resend the last message, it may result in a duplicate message being sent to the address. If each message was an order or a trade then this could result in the order being fulfilled twice or the trade being doubly booked, which is an undesirable situation.

Sending the message(s) in a transaction is also not the correct solution. If the server or connection fails while the transaction commit is being processed, it is indeterminate whether the transaction was successfully committed or not.

To solve these issues HornetQ provides automatic duplicate messages detection for messages sent to addresses.

37.1. Using Duplicate Detection for Message Sending

Enable duplicate message detection for sent messages: set a special property on the message to a uniquely created value. When the target server receives the message it will check whether that property is set. If it is, the target server will check its memory cache whether a message with the value of the header has already been received. If it has previously received a message with the same value it will ignore the message.

Note

Using duplicate detection to move messages between nodes can give the same *once and only once* delivery guarantees as when an XA transaction is used to consume messages from source and send them to the target, but with less overhead and much easier configuration than using XA.

If sending messages in a transaction it is not necessary to set the property for *every* message sent in that transaction, set it once in the transaction. If the server detects a duplicate message for any message in the transaction, it will ignore the entire transaction.

The name of the property set is given by the value of `org.hornetq.api.core.HDR_DUPLICATE_DETECTION_ID`, which is `_HQ_DUPL_ID`.

The value of the property can be of type `byte[]` or `SimpleString` if using the core API. If using JMS it must be a `String`, and its value should be unique. An easy way of generating a unique ID is by generating a UUID.

An example of setting the property using the core API follows:

```
...
ClientMessage message = session.createMessage(true);
SimpleString myUniqueID = "This is my unique id";   // Could use a UUID for this
message.setStringProperty(HDR_DUPLICATE_DETECTION_ID, myUniqueID);
...
```

And here's an example using the JMS API:

```
...
Message jmsMessage = session.createMessage();
String myUniqueID = "This is my unique id";   // Could use a UUID for this
message.setStringProperty(HDR_DUPLICATE_DETECTION_ID.toString(), myUniqueID);
...
```

37.2. Configuring the Duplicate ID Cache

The server maintains caches of received values of the `org.hornetq.core.message.impl.HDR_DUPLICATE_DETECTION_ID` property sent to each address. Each address has its own distinct cache.

The cache is a circular fixed size cache. If the cache has a maximum size of **n** elements, then the **n + 1**th id stored will overwrite the **0**th element in the cache.

The maximum size of the cache is configured by the parameter `id-cache-size` in **`hornetq-configuration.xml`**, the default value is **`2000`** elements.

The caches can also be configured to persist to disk or not. This is configured by the parameter `persist-id-cache`, also in **`hornetq-configuration.xml`**. If this is set to **`true`** then each id will be persisted to permanent storage as it is received. The default value for this parameter is **`true`**.

Note

When choosing a size for the duplicate id cache, be sure to set it to a size large enough to prevent the previously sent messages from being overwritten.

37.3. Duplicate Detection and Bridges

Core bridges can be configured to automatically add a unique duplicate id value (if there isn't already one in the message) before forwarding the message to it's target. This ensures that if the target server crashes or the connection is interrupted and the bridge resends the message, then if it has already been received by the target server, it will be ignored.

To configure a core bridge to add the duplicate id header, simply set the *use-duplicate-detection* to **true** when configuring a bridge in **hornetq-configuration.xml**.

The default value for this parameter is **true**.

For more information on core bridges and how to configure them, refer to *Chapter 36, Core Bridges*.

37.4. Duplicate Detection and Cluster Connections

Cluster connections internally use core bridges to move messages reliable between nodes of the cluster. Consequently cluster connections can also be configured to insert the duplicate id header for each message moved using internal bridges.

To configure a cluster connection to add the duplicate id header, simply set the *use-duplicate-detection* to **true** when configuring a cluster connection in **hornetq-configuration.xml**.

The default value for this parameter is **true**.

For more information on cluster connections and how to configure them, refer to *Chapter 38, Clusters*.

37.5. Duplicate Detection and Paging

HornetQ also uses duplicate detection when paging messages to storage. When a message is depaged from storage and server failure occurs, the message is not depaged more than once which could result in duplicate delivery.

For more information on paging and how to configure it, refer to *Chapter 24, Paging*.

Clusters

38.1. Clusters Overview

HornetQ clusters allow groups of HornetQ servers to be grouped together in order to share message processing load. Each active node in the cluster is an active HornetQ server which manages its own messages and handles its own connections. A server must be configured to be clustered, you will need to set the **`clustered`** element in the **`hornetq-configuration.xml`** configuration file to **`true`**, this is **`false`** by default.

The cluster is formed by each node declaring *cluster connections* to other nodes in the core configuration file **`hornetq-configuration.xml`**. When a node forms a cluster connection to another node, internally it creates a *core bridge* (as described in *Chapter 36, Core Bridges*) connection between it and the other node, this is done transparently behind the scenes - you don't have to declare an explicit bridge for each node. These cluster connections allow messages to flow between the nodes of the cluster to balance load.

Nodes can be connected together to form a cluster in many different topologies, more of the common topologies are discussed later in this chapter.

Also discussed is client side load balancing, where client connections can be balanced across the nodes of the cluster. Message redistribution where HornetQ will redistribute messages between nodes to avoid starvation is also covered.

Another important part of clustering is *server discovery* where servers can broadcast their connection details so clients or other servers can connect to them with the minimum of configuration.

Warning - Cluster Isolation

If you start a JBoss Enterprise Application Platform instance with a HornetQ server bound to 'localhost', the HornetQ instance could form a cluster with another HornetQ instance on the same network. Binding to localhost does not provide cluster isolation for HornetQ servers.

To correctly isolate clusters, refer to *Section 38.2.1, "Broadcast Groups"* to correctly configure the broadcast and discovery addresses of each server.

38.2. Server discovery

Server discovery is a mechanism by which servers can propagate their connection details to:

- Messaging clients. A messaging client wants to be able to connect to the servers of the cluster without having specific knowledge of which servers in the cluster are up at any one time.
- Other servers. Servers in a cluster want to be able to create cluster connections to each other without having prior knowledge of all the other servers in the cluster.

Server discovery uses UDP (User Datagram Protocol) multicast to broadcast server connection settings. If UDP is disabled on your network you will not be able to use this, and will have to specify servers explicitly when setting up a cluster or using a messaging client.

38.2.1. Broadcast Groups

A broadcast group is the means by which a server broadcasts connectors over the network. A connector defines a way in which a client (or other server) can make connections to the server. For more information on what a connector is, please see *Chapter 16, Configuring the Transport*.

The broadcast group takes a set of connector pairs, each connector pair contains connection settings for a live and (optional) backup server and broadcasts them on the network. It also defines the UDP address and port settings.

Broadcast groups are defined in the server configuration file **`hornetq-configuration.xml`**. There can be many broadcast groups per HornetQ server. All broadcast groups must be defined in a **`broadcast-groups`** element.

Let's take a look at an example broadcast group from **`hornetq-configuration.xml`**:

```
<broadcast-groups>
   <broadcast-group name="my-broadcast-group">
      <local-bind-address>172.16.9.3</local-bind-address>
      <local-bind-port>5432</local-bind-port>
      <group-address>231.7.7.7</group-address>
      <group-port>9876</group-port>
      <broadcast-period>2000</broadcast-period>
      <connector-ref connector-name="netty-connector"
        backup-connector-name="backup-connector"/>
   </broadcast-group>
</broadcast-groups>
```

Some of the broadcast group parameters are optional and the defaults will normally be used, but all are specified in the above example for clarity. Each are covered below.

Broadcast Group Parameters

`name attribute`
: Each broadcast group in the server must have a unique name in the server.

`local-bind-address`
: This is the local bind address that the datagram socket is bound to. If you have multiple network interfaces on your server, you would specify which one you wish to use for broadcasts by setting this property. If this property is not specified then the socket will be bound to the wildcard address, an IP address chosen by the kernel.

`local-bind-port`
: If you want to specify a local port to which the datagram socket is bound you can specify it here. Normally you would just use the default value of **`-1`** which signifies that an anonymous port should be used. This parameter is always specified in conjunction with **`local-bind-address`**.

`group-address`
: This is the multicast address to which the data will be broadcast. It is a class D IP address in the range **`224.0.0.0`** to **`239.255.255.255`**, inclusive. The address **`224.0.0.0`** is reserved and is not available for use. This parameter is mandatory.

`group-port`
: This is the UDP port number used for broadcasting. This parameter is mandatory.

`broadcast-period`
: This is the period in milliseconds between consecutive broadcasts. This parameter is optional, the default value is **`2000`** milliseconds.

`connector-ref`
: This specifies the connector and optional backup connector that will be broadcast (see *Chapter 16, Configuring the Transport* for more information on connectors). The connector to be broadcasted is specified by the **`connector-name`** attribute, and the backup connector to be broadcasted is specified by the **`backup-connector`** attribute. The **`backup-connector`** attribute is optional.

38.2.2. Discovery Groups

While the broadcast group defines how connector information is broadcasted from a server, a discovery group defines how connector information is received from a multicast address.

A discovery group maintains a list of connector pairs - one for each broadcast by a different server. As it receives broadcasts on the multicast group address from a particular server it updates its entry in the list for that server.

If it has not received a broadcast from a particular server for a length of time it will remove that server's entry from its list.

Discovery groups are used in two places in HornetQ:

- By cluster connections so they know what other servers in the cluster they should make connections to.
- By messaging clients so they can discovery what servers in the cluster they can connect to.

Although a discovery group will always accept broadcasts, its current list of available live and backup servers is only ever used when an initial connection is made, from then server discovery is done over the normal HornetQ connections.

38.2.3. Defining Discovery Groups on the Server

For cluster connections, discovery groups are defined in the server side configuration file **`hornetq-configuration.xml`**. All discovery groups must be defined inside a **`discovery-groups`** element. There can be many discovery groups defined by HornetQ server. Let's look at an example:

```
<discovery-groups>
   <discovery-group name="my-discovery-group">
      <local-bind-address>172.16.9.7</local-bind-address>
      <group-address>231.7.7.7</group-address>
      <group-port>9876</group-port>
      <refresh-timeout>10000</refresh-timeout>
   </discovery-group>
</discovery-groups>
```

Each parameter of the discovery group is considered as follows:

Discovery Group Parameters

`name`
: Each discovery group must have a unique name per server.

`local-bind-address`
: If you are running with multiple network interfaces on the same machine, you may want to specify that the discovery group listens only only a specific interface. To do this you can specify the interface address with this parameter. This parameter is optional.

`group-address`
: This is the multicast IP address of the group to listen on. It should match the **`group-address`** in the broadcast group that you wish to listen from. This parameter is mandatory.

`group-port`
: This is the UDP port of the multicast group. It should match the **`group-port`** in the broadcast group that you wish to listen from. This parameter is mandatory.

`refresh-timeout`
: This is the period the discovery group waits after receiving the last broadcast from a particular server before removing that servers connector pair entry from its list. You would normally set this to a value significantly higher than the **`broadcast-period`** on the broadcast group otherwise servers might intermittently disappear from the list even though they are still broadcasting due to slight differences in timing. This parameter is optional, the default value is **`10000`** milliseconds (10 seconds).

38.2.4. Discovery Groups on the Client Side

Let's discuss how to configure a HornetQ client to use discovery to discover a list of servers to which it can connect. The way to do this differs depending on whether you're using JMS or the core API.

38.2.4.1. Configuring client discovery using JMS

If you're using JMS and you're also using the JMS Service on the server to load your JMS connection factory instances into JNDI, then you can specify which discovery group to use for your JMS connection factory in the server side xml configuration **`hornetq-jms.xml`**. Let's take a look at an example:

```
<connection-factory name="ConnectionFactory">
   <discovery-group-ref discovery-group-name="my-discovery-group"/>
    <entries>
       <entry name="ConnectionFactory"/>
    </entries>
</connection-factory>
```

The element **`discovery-group-ref`** specifies the name of a discovery group defined in **`hornetq-configuration.xml`**.

When this connection factory is downloaded from JNDI by a client application and JMS connections are created from it, those connections will be load-balanced across the list of servers that the discovery group maintains by listening on the multicast address specified in the discovery group configuration.

If you're using JMS, but you're not using JNDI to lookup a connection factory - you're instantiating the JMS connection factory directly then you can specify the discovery group parameters directly when creating the JMS connection factory. Here's an example:

```
final String groupAddress = "231.7.7.7";

final int groupPort = 9876;

ConnectionFactory jmsConnectionFactory =
   HornetQJMSClient.createConnectionFactory(groupAddress, groupPort);

Connection jmsConnection1 = jmsConnectionFactory.createConnection();

Connection jmsConnection2 = jmsConnectionFactory.createConnection();
```

The **refresh-timeout** can be set directly on the connection factory by using the setter method **setDiscoveryRefreshTimeout()** if you want to change the default value.

There is also a further parameter settable on the connection factory using the setter method **setDiscoveryInitialWaitTimeout()**. If the connection factory is used immediately after creation then it may not have had enough time to received broadcasts from all the nodes in the cluster. On first usage, the connection factory will make sure it waits this long since creation before creating the first connection. The default value for this parameter is **10000** milliseconds.

38.2.4.2. Configuring client discovery using Core

If you're using the core API to directly instantiate **ClientSessionFactory** instances, then you can specify the discovery group parameters directly when creating the session factory. Here's an example:

```
final String groupAddress = "231.7.7.7";
final int groupPort = 9876;
SessionFactory factory = HornetQClient.createClientSessionFactory
  (groupAddress, groupPort);
ClientSession session1 = factory.createClientSession(...);
ClientSession session2 = factory.createClientSession(...);
```

The **refresh-timeout** can be set directly on the session factory by using the setter method **setDiscoveryRefreshTimeout()** if you want to change the default value.

There is also a further parameter settable on the session factory using the setter method **setDiscoveryInitialWaitTimeout()**. If the session factory is used immediately after creation then it may not have had enough time to received broadcasts from all the nodes in the cluster. On first usage, the session factory will make sure it waits this long since creation before creating the first session. The default value for this parameter is **10000** milliseconds.

38.3. Server-Side Message Load Balancing

If cluster connections are defined between nodes of a cluster, then HornetQ will load balance messages arriving at a particular node from a client.

Let's take a simple example of a cluster of four nodes A, B, C, and D arranged in a *symmetric cluster* (described in *Section 38.7.1, "Symmetric cluster"*). A queue called **OrderQueue** is deployed on each node of the cluster.

A client Ca is connected to node A, sending orders to the server. Also, order processor clients Pa, Pb, Pc, and Pd are connected to each of the nodes A, B, C, D. If no cluster connection was defined on node A, as order messages arrive on node A they will all end up in the **OrderQueue** on node A, so will only get consumed by the order processor client attached to node A, Pa.

If a cluster connection on node A is defined, as ordered messages arrive on node A, they are distributed in a round-robin fashion between all the nodes of the cluster, instead of all of them going into the local **OrderQueue** instance. The messages are forwarded from the receiving node to other nodes of the cluster. This is all done on the server side, the client maintains a single connection to node A.

For example, messages arriving on node A might be distributed in the following order between the nodes: B, D, C, A, B, D, C, A, B, D. The exact order depends on the order the nodes started up, but the algorithm used is round robin.

HornetQ cluster connections can be configured to always blindly load balance messages in a round robin fashion irrespective of whether there are any matching consumers on other nodes, but they can be a bit cleverer than that and also be configured to only distribute to other nodes if they have

matching consumers. Both of these cases will be covered in turn with some examples. Firstly, configuring cluster connections in general will be covered.

38.3.1. Configuring Cluster Connections

Cluster connections group servers into clusters so that messages can be load balanced between the nodes of the cluster. Typical cluster connections are defined in **`hornetq-configuration.xml`** inside a **`cluster-connection`** element. There can be zero or more cluster connections defined per HornetQ server.

```
<cluster-connections>
   <cluster-connection name="my-cluster">
      <address>jms</address>
      <retry-interval>500</retry-interval>
      <use-duplicate-detection>true</use-duplicate-detection>
      <forward-when-no-consumers>false</forward-when-no-consumers>
      <max-hops>1</max-hops>
      <discovery-group-ref discovery-group-name="my-discovery-group"/>
   </cluster-connection>
</cluster-connections>
```

In the above cluster connection all parameters have been explicitly specified. In practice you might use the defaults for some.

- **`address`**. Each cluster connection only applies to messages sent to an address that starts with this value.

  In this case, this cluster connection will load balance messages sent to address that start with **`jms`**. This cluster connection, applies to all JMS queue and topic subscriptions since they map to core queues that start with the substring "jms".

  The address can be any value and you can have many cluster connections with different values of **`address`**, simultaneously balancing messages for those addresses, potentially to different clusters of servers. By having multiple cluster connections on different addresses a single HornetQ Server can effectively take part in multiple clusters simultaneously.

  Be careful not to have multiple cluster connections with overlapping values of **`address`**, e.g. "europe" and "europe.news" since this could result in the same messages being distributed between more than one cluster connection, possibly resulting in duplicate deliveries.

  This parameter is mandatory.

- **`retry-interval`**. Internally, cluster connections cause bridges to be created between the nodes of the cluster. If the cluster connection is created and the target node has not been started, or say, is being rebooted, then the cluster connections from other nodes will retry connecting to the target until it comes back up, in the same way as a bridge does.

  This parameter determines the interval in milliseconds between retry attempts. It has the same meaning as the **`retry-interval`** on a bridge (as described in *Chapter 36, Core Bridges*).

  This parameter is optional and its default value is **`500`** milliseconds.

- **`use-duplicate-detection`**. Internally cluster connections use bridges to link the nodes, and bridges can be configured to add a duplicate id property in each message that is forwarded. If the target node of the bridge crashes and then recovers, messages might be resent from the source node. By enabling duplicate detection any duplicate messages will be filtered out and ignored on receipt at the target node.

This parameter has the same meaning as **`use-duplicate-detection`** on a bridge. For more information on duplicate detection, please see *Chapter 37, Duplicate Message Detection*.

This parameter is optional and has a default value of **`true`**.

- **`forward-when-no-consumers`**. This parameter determines whether messages will be distributed round robin between other nodes of the cluster *irrespective* of whether there are matching or indeed any consumers on other nodes.

  If this is set to **`true`** then each incoming message will be processed in a round robin style even though the same queues on the other nodes of the cluster may have no consumers at all, or they may have consumers that have non matching message filters (selectors). Note that HornetQ will *not* forward messages to other nodes if there are no *queues* of the same name on the other nodes, even if this parameter is set to **`true`**.

  If this is set to **`false`** then HornetQ will only forward messages to other nodes of the cluster if the address to which they are being forwarded has queues which have consumers, and if those consumers have message filters (selectors) at least one of those selectors must match the message.

  This parameter is optional, the default value is **`false`**.

- **`max-hops`**. When a cluster connection decides the set of nodes to which it might load balance a message, those nodes do not have to be directly connected to it via a cluster connection. HornetQ can be configured to also load balance messages to nodes which might be connected to it only indirectly with other HornetQ servers as intermediates in a chain.

  This allows HornetQ to be configured in more complex topologies and still provide message load balancing. This is covered later in this chapter.

  The default value for this parameter is **`1`**, which means messages are only load balanced to other HornetQ serves which are directly connected to this server. This parameter is optional.

- **`discovery-group-ref`**. This parameter determines which discovery group is used to obtain the list of other servers in the cluster that this cluster connection will make connections to.

38.3.2. Cluster User Credentials

When creating connections between nodes of a cluster to form a cluster connection, HornetQ uses a cluster user and cluster password which is defined in **`hornetq-configuration.xml`**:

```
<cluster-user>HORNETQ.CLUSTER.ADMIN.USER</cluster-user>
<cluster-password>CHANGE ME!!</cluster-password>
```

Warning

It is imperative that these values are changed from their default, or remote clients will be able to make connections to the server using the default values. If they are not changed from the default, HornetQ will detect this and pester you with a warning on every start-up.

38.4. Client-Side Load balancing

With HornetQ client-side load balancing, subsequent sessions created using a single session factory can be connected to different nodes of the cluster. This allows sessions to spread smoothly across the nodes of a cluster and not be "clumped" on any particular node.

The load balancing policy to be used by the client factory is configurable. HornetQ provides two out-of-the-box load balancing policies and you can also implement your own and use that.

The out-of-the-box policies are:

- Round Robin. With this policy the first node is chosen randomly then each subsequent node is chosen sequentially in the same order.

  For example nodes might be chosen in the order B, C, D, A, B, C, D, A, B or D, A, B, C, A, B, C, D, A or C, D, A, B, C, D, A, B, C, D, A.

- Random. With this policy each node is chosen randomly.

It is possible to implement your own policy by implementing the interface **`org.hornetq.api.core.client.loadbalance.ConnectionLoadBalancingPolicy`**

Specifying which load balancing policy to use differs whether you are using JMS or the core API. If you don't specify a policy then the default will be used which is **`org.hornetq.api.core.client.loadbalance.RoundRobinConnectionLoadBalancingPolicy`**.

If you're using JMS, and you're using JNDI on the server to put your JMS connection factories into JNDI, then you can specify the load balancing policy directly in the **`hornetq-jms.xml`** configuration file on the server as follows:

```
<connection-factory name="ConnectionFactory">
   <discovery-group-ref discovery-group-name="my-discovery-group"/>
   <entries>
      <entry name="ConnectionFactory"/>
   </entries>
   <ha>true</ha>
   <connection-load-balancing-policy-class-name>
    org.hornetq.api.core.client.loadbalance.RandomConnectionLoadBalancingPolicy
   </connection-load-balancing-policy-class-name>
</connection-factory>
```

The above example would deploy a JMS connection factory that uses the random connection load balancing policy.

If you're using JMS but you're instantiating your connection factory directly on the client side then you can set the load balancing policy using the setter on the **`HornetQConnectionFactory`** before using it:

```
ConnectionFactory jmsConnectionFactory = HornetQJMSClient.createConnectionFactory(...);
jmsConnectionFactory.setLoadBalancingPolicyClassName("com.acme.MyLoadBalancingPolicy");
```

If you're using the core API, you can set the load balancing policy directly on the **`ClientSessionFactory`** instance you are using:

```
ClientSessionFactory factory = HornetQClient.createClientSessionFactory(...);
factory.setLoadBalancingPolicyClassName("com.acme.MyLoadBalancingPolicy");
```

The set of servers over which the factory load balances can be determined in one of two ways:

- Specifying servers explicitly
- Using discovery.

38.5. Specifying Members of a Cluster Explicitly

Sometimes UDP is not enabled on a network so it's not possible to use UDP server discovery for clients to discover the list of servers in the cluster, or for servers to discover what other servers are in the cluster.

In this case, the list of servers in the cluster can be specified explicitly on each node and on the client side. This is done as follows:

38.5.1. Specify List of Servers on the Client Side

This differs depending on whether you're using JMS or the Core API

38.5.1.1. Specifying List of Servers using JMS

If using JMS and the JMS Service to load your JMS connection factory instances directly into JNDI on the server, then you can specify the list of servers in the server side configuration file **`hornetq-jms.xml`**. Let's take a look at an example:

```
<connection-factory name="ConnectionFactory">
   <connectors>
      <connector-ref connector-name="my-connector1"
       backup-connector-name="my-backup-connector1"/>
      <connector-ref connector-name="my-connector2"
       backup-connector-name="my-backup-connector2"/>
      <connector-ref connector-name="my-connector3"
       backup-connector-name="my-backup-connector3"/>
   </connectors>
   <entries>
      <entry name="ConnectionFactory"/>
   </entries>
</connection-factory>
```

The **`connection-factory`** element can contain zero or more **`connector-ref`** elements, each one of which specifies a **`connector-name`** attribute and an optional **`backup-connector-name`** attribute. The **`connector-name`** attribute references a connector defined in **`hornetq-configuration.xml`** which will be used as a live connector. The **`backup-connector-name`** is optional, and if specified it also references a connector defined in **`hornetq-configuration.xml`**. For more information on connectors please see *Chapter 16, Configuring the Transport*.

The connection factory thus maintains a list of [connector, backup connector] pairs, these pairs are then used by the client connection load balancing policy on the client side when creating connections to the cluster.

If you're using JMS but you're not using JNDI then you can also specify the list of [connector, backup connector] pairs directly when instantiating the **`HornetQConnectionFactory`**, here's an example:

```
List<Pair<TransportConfiguration, TransportConfiguration>> serverList =
        new ArrayList<Pair<TransportConfiguration, TransportConfiguration>>();

serverList.add(new Pair<TransportConfiguration,
        TransportConfiguration>(liveTC0, backupTC0));
serverList.add(new Pair<TransportConfiguration,
        TransportConfiguration>(liveTC1, backupTC1));
```

```
serverList.add(new Pair<TransportConfiguration,
        TransportConfiguration>(liveTC2, backupTC2));

ConnectionFactory jmsConnectionFactory =
 HornetQJMSClient.createConnectionFactory(serverList);

Connection jmsConnection1 = jmsConnectionFactory.createConnection();

Connection jmsConnection2 = jmsConnectionFactory.createConnection();
```

The above snippet creates a list of pairs of **`TransportConfiguration`** objects. Each **`TransportConfiguration`** object contains knowledge of how to make a connection to a specific server.

Create a **`HornetQConnectionFactory`** instance, passing the list of servers in the constructor. Any connections subsequently created by this factory will create connections according to the client connection load balancing policy applied to that list of servers.

38.5.1.2. Specifying List of Servers using the Core API

If using the core API, also specify the list of servers directly when creating the **`ClientSessionFactory`** instance as in the following example:

```
List<Pair<TransportConfiguration, TransportConfiguration>> serverList =
        new ArrayList<Pair<TransportConfiguration, TransportConfiguration>>();

serverList.add(new Pair<TransportConfiguration,
        TransportConfiguration>(liveTC0, backupTC0));
serverList.add(new Pair<TransportConfiguration,
        TransportConfiguration>(liveTC1, backupTC1));
serverList.add(new Pair<TransportConfiguration,
        TransportConfiguration>(liveTC2, backupTC2));

ClientSessionFactory factory = HornetQClient.createClientSessionFactory(serverList);

ClientSession session1 = factory.createClientSession(...);

ClientSession session2 = factory.createClientSession(...);
```

The above snippet creates a list of pairs of **`TransportConfiguration`** objects. Each **`TransportConfiguration`** object contains knowledge of how to make a connection to a specific server. For more information on this, refer to *Chapter 16, Configuring the Transport*.

A **`ClientSessionFactoryImpl`** instance is then created passing the list of servers in the constructor. Any sessions subsequently created by this factory will create sessions according to the client connection load balancing policy applied to that list of servers.

38.5.2. Specifying List of Servers to form a Cluster

An example is provided where each cluster connection is defined for a symmetric cluster, not using discovery for each node to discover its neighbors, but configuring each cluster connection to have explicit knowledge of all the other nodes in the cluster.

An example cluster connection definition follows:

```
<cluster-connections>
    <cluster-connection name="my-explicit-cluster">
        <address>jms</address>
        <connector-ref connector-name="my-connector1"
            backup-connector-name="my-backup-connector1"/>
```

```
        <connector-ref connector-name="my-connector2"
            backup-connector-name="my-backup-connector2"/>
        <connector-ref connector-name="my-connector3"
            backup-connector-name="my-backup-connector3"/>
    </cluster-connection>
</cluster-connections>
```

The **`cluster-connection`** element can contain zero or more **`connector-ref`** elements, each one of which specifies a **`connector-name`** attribute and an optional **`backup-connector-name`** attribute. The **`connector-name`** attribute references a connector defined in **`hornetq-configuration.xml`** which will be used as a live connector. The **`backup-connector-name`** is optional, and if specified it also references a connector defined in **`hornetq-configuration.xml`**. For more information on connectors please see *Chapter 16, Configuring the Transport*.

Note

Due to a limitation in HornetQ 2.2, failover is not supported for clusters defined using a static set of nodes. To support failover over cluster nodes, they must be configured to use a discovery group.

38.6. Message Redistribution

Another important part of clustering is message redistribution. Earlier, it was demonstrated how server side message load balancing round robins or directs messages to all nodes across the cluster. If **`forward-when-no-consumers`** is false, then messages will not be forwarded to nodes which don't have matching consumers, this is great and ensures that messages don't arrive on a queue which has no consumers to consume them, however there is a situation it doesn't solve: What happens if the consumers on a queue close after the messages have been sent to the node? If there are no consumers on the queue the message will not get consumed and a *starvation* situation will be created.

Message redistribution addresses this and HornetQ can be configured to automatically *redistribute* messages from queues which have no consumers back to other nodes in the cluster which do have matching consumers.

Message redistribution can be configured to kick in immediately after the last consumer on a queue is closed, or to wait for a configurable delay after the last consumer on a queue is closed before redistributing. By default, message redistribution is disabled.

Message redistribution can be configured on a per address basis, by specifying the redistribution delay in the address settings, for more information on configuring address settings, refer to *Chapter 25, Queue Attributes*.

An address settings snippet from **`hornetq-configuration.xml`** showing how message redistribution is enabled for a set of queues follows:

```
<address-settings>
   <address-setting match="jms.#">
      <redistribution-delay>0</redistribution-delay>
   </address-setting>
</address-settings>
```

The above **`address-settings`** block would set a **`redistribution-delay`** of **`0`** for any queue which is bound to an address that starts with "jms.". All JMS queues and topic subscriptions are bound to addresses that start with "jms.", so the above would enable instant (no delay) redistribution for all JMS queues and topic subscriptions.

The attribute **match** can be an exact match or it can be a string that conforms to the HornetQ wildcard syntax (described in *Chapter 13, Understanding the HornetQ Wildcard Syntax*).

The element **redistribution-delay** defines the delay in milliseconds after the last consumer is closed on a queue before redistributing messages from that queue to other nodes of the cluster which do have matching consumers. A delay of zero means the messages will be immediately redistributed. A value of **-1** signifies that messages will never be redistributed. The default value is **-1**.

It often makes sense to introduce a delay before redistributing as it's a common case that a consumer closes but another one quickly is created on the same queue, in such a case you probably don't want to redistribute immediately since the new consumer will arrive shortly.

38.7. Cluster topologies

HornetQ clusters can be connected together in many different topologies, let's consider the two most common ones here:

38.7.1. Symmetric cluster

A symmetric cluster is probably the most common cluster topology, and you'll be familiar with if you've had experience of JBoss Application Server clustering.

With a symmetric cluster every node in the cluster is connected to every other node in the cluster. In other words every node in the cluster is no more than one hop away from every other node.

To form a symmetric cluster every node in the cluster defines a cluster connection with the attribute **max-hops** set to **1**. Typically the cluster connection will use server discovery in order to know what other servers in the cluster it should connect to, although it is possible to explicitly define each target server too in the cluster connection if, for example, UDP is not available on your network.

With a symmetric cluster each node knows about all the queues that exist on all the other nodes and what consumers they have. With this knowledge it can determine how to load balance and redistribute messages around the nodes.

38.7.2. Chain cluster

With a chain cluster, each node in the cluster is not connected to every node in the cluster directly, instead the nodes form a chain with a node on each end of the chain and all other nodes just connecting to the previous and next nodes in the chain.

An example of this would be a three node chain consisting of nodes A, B and C. Node A is hosted in one network and has many producer clients connected to it sending order messages. Due to corporate policy, the order consumer clients need to be hosted in a different network, and that network is only accessible via a third network. In this setup node B acts as a mediator with no producers or consumers on it. Any messages arriving on node A will be forwarded to node B, which will in turn forward them to node C where they can get consumed. Node A does not need to directly connect to C, but all the nodes can still act as a part of the cluster.

To set up a cluster in this way, node A would define a cluster connection that connects to node B, and node B would define a cluster connection that connects to node C. In this case, cluster connections are only desired in one direction since messages are only moving from node A->B->C and never from C->B->A.

For this topology, set **max-hops** to **2**. With a value of **2** the knowledge of what queues and consumers that exist on node C would be propagated from node C to node B to node A. Node A would then know

to distribute messages to node B when they arrive, even though node B has no consumers itself, it would know that a further hop away is node C which does have consumers.

High Availability and Failover

High availability is defined as the ability for the system to continue functioning after failure of one or more of the servers.

A part of high availability is *failover* which is defined as the *ability for client connections to migrate from one server to another in the event of server failure so that client applications can continue to operate*.

39.1. Live - Backup Pairs

HornetQ allows pairs of servers to be linked together as *live - backup* pairs. In this release there is a single backup server for each live server. A backup server is owned by only one live server. Backup servers are not operational until failover occurs.

Before failover, only the live server is serving the HornetQ clients while the backup servers remain passive or awaiting to become a backup server. When a live server crashes or is brought down in the correct mode, the backup server currently in passive mode will become live and another backup server will become passive. If a live server restarts after a failover then it will have priority and be the next server to become live when the current live server goes down, if the current live server is configured to allow automatic failback then it will detect the live server coming back up and automatically stop.

39.1.1. HA modes

HornetQ provides only *shared store* in this release. *Replication* will be available in subsequent releases.

Note

Only persistent message data will survive failover. Non-persistent message data is lost after failover occurs.

39.1.1.1. Data Replication

Replication will be available in subsequent HornetQ releases.

39.1.2. Shared Store

When using a shared store, both live and backup servers share the *same* entire data directory using a shared file system. This means the paging directory, journal directory, large messages and binding journal.

When failover occurs and the backup server takes over, it will load the persistent storage from the shared file system and clients can connect to it.

This style of high availability differs from data replication in that it requires a shared file system which is accessible by both the live and backup nodes. Typically this will be some kind of high performance Storage Area Network (SAN). It is not recommend to use Network Attached Storage (NAS), e.g. NFS mounts to store any shared journal (NFS is slow).

The advantage of shared-store high availability is that no replication occurs between the live and backup nodes, this means it does not suffer any performance penalties due to the overhead of replication during normal operation.

The disadvantage of shared store replication is that it requires a shared file system, and when the backup server activates it needs to load the journal from the shared store which can take some time depending on the amount of data in the store.

If the highest performance during normal operation is required and there is access to a fast SAN, and a slightly slower failover is acceptable (depending on amount of data), shared store high availability is recommended.

39.1.2.1. Configuration

To configure the live and backup server to share their store, configure both **`hornetq-configuration.xml`**:

```
<shared-store>true</shared-store>
```

Additionally, the backup server must be flagged explicitly as a backup:

```
<backup>true</backup>
```

In order for live - backup pairs to operate properly with a shared store, both servers must have configured the location of journal directory to point to the *same shared location* (as explained in *Section 15.3, "Configuring the message journal"*)

Also each node, live and backups, will need to have a cluster connection defined even if not part of a cluster. The Cluster Connection info defines how backup servers announce there presence to it's live server or any other nodes in the cluster. refer to *Chapter 38, Clusters* for details on how this is done.

39.1.2.2. Failing Back to live Server

After a live server has failed and a backup taken has taken over its duties, you may want to restart the live server and have clients fail back. To do this simply restart the original live server and kill the new live server. You can do this by killing the process itself or just waiting for the server to crash naturally

It is also possible to cause failover to occur on normal server shutdown, to enable this set the following property to true in the **`hornetq-configuration.xml`** configuration file like so:

```
<failover-on-shutdown>true</failover-on-shutdown>
```

By default this is set to false, if by some chance you have set this to false but still want to stop the server normally and cause failover then you can do this by using the management API as explained at ...

You can also force the new live server to shutdown when the old live server comes back up allowing the original live server to take over automatically by setting the following property in the **`hornetq-configuration.xml`** configuration file as follows:

```
<allow-failback>true</allow-failback>
```

39.2. Failover Modes

HornetQ defines two types of client failover:

- Automatic client failover
- Application-level client failover

HornetQ also provides 100% transparent automatic reattachment of connections to the same server (e.g. in case of transient network problems). This is similar to failover, except it's reconnecting to the same server and is discussed in *Chapter 34, Client Reconnection and Session Reattachment*

During failover, if the client has consumers on any non persistent or temporary queues, those queues will be automatically recreated during failover on the backup node, since the backup node will not have any knowledge of non persistent queues.

39.2.1. Automatic Client Failover

HornetQ clients can be configured with knowledge of live and backup servers, so that in event of connection failure at the client - live server connection, the client will detect this and reconnect to the backup server. The backup server will then automatically recreate any sessions and consumers that existed on each connection before failover, thus saving the user from having to hand-code manual reconnection logic.

HornetQ clients detect connection failure when it has not received packets from the server within the time given by **`client-failure-check-period`** as explained in section *Chapter 17, Detecting Dead Connections*. If the client does not receive data in good time, it will assume the connection has failed and attempt failover.

HornetQ clients can be configured with the list of live-backup server pairs in a number of different ways. They can be configured explicitly or probably the most common way of doing this is to use *server discovery* for the client to automatically discover the list. For full details on how to configure server discovery, please see *Section 38.2, "Server discovery"*. Alternatively, the clients can explicitly specifies pairs of live-backup server as explained in *Section 38.5.2, "Specifying List of Servers to form a Cluster"*.

To enable automatic client failover, the client must be configured to allow non-zero reconnection attempts (as explained in *Chapter 34, Client Reconnection and Session Reattachment*).

Sometimes it is desireable for a client to failover onto a backup server even if the live server is just cleanly shutdown rather than having crashed or the connection failed. To configure this set the property **`FailoverOnServerShutdown`** to true either on the **`HornetQConnectionFactory`** if using JMS or in the **`hornetq-jms.xml (failover-on-server-shutdown property)`** file when defining the connection factory, or if using core by setting the property directly on the **`ClientSessionFactoryImpl`** instance after creation. The default value for this property is **`false`**, this means that by default *HornetQ clients will not failover to a backup server if the live server is simply shutdown cleanly.*

Note

By default, cleanly shutting down the server will not trigger failover on the client.

Using CTRL-C on a HornetQ server or JBoss AS instance causes the server to cleanly shut down, so will not trigger failover on the client.

For the client to failover when its server is cleanly shutdown, set the property **`FailoverOnServerShutdown`** to true

By default failover will only occur after at least one connection has been made to the live server. In other words, by default, failover will not occur if the client fails to make an initial connection to the live server - in this case it will simply retry connecting to the live server according to the reconnect-attempts property and fail after this number of attempts.

In some cases, you may want the client to automatically try the backup server it fails to make an initial connection to the live server. In this case, set the property **`FailoverOnInitialConnection`**, or **`failover-on-initial-connection`** in xml, on the **`ClientSessionFactoryImpl`** or **`HornetQConnectionFactory`**. The default value for this parameter is **`false`**.

For examples of automatic failover with transacted and non-transacted JMS sessions, please see *Section 11.1.56, "Transaction Failover With Data Replication"* and *Section 11.1.32, "Non-Transaction Failover With Server Data Replication"*.

39.2.1.1. A Note on Server Replication

HornetQ does not replicate full server state between live and backup servers. When the new session is automatically recreated on the backup it will not have any knowledge of messages already sent or acknowledged in that session. Any in-flight sends or acknowledgments at the time of failover might also be lost.

Theoretically, by replicating full server state, a 100% transparent seamless failover could be provided, which would avoid any lost messages or acknowledgments. However this comes at a great cost: replicating the full server state (including the queues, session, etc.). It would require replication of the entire server state machine; every operation on the live server would have to replicated on the replica server(s) in the exact same global order to ensure a consistent replica state. This is extremely hard to do in a performant and scalable way, especially when one considers that multiple threads are changing the live server state concurrently.

It is possible to provide full state machine replication using techniques such as *virtual synchrony*, but this does not scale well and effectively serializes all operations to a single thread, dramatically reducing concurrency.

Other techniques for multi-threaded active replication exist such as replicating lock states or replicating thread scheduling but this is very hard to achieve at a Java level.

Consequently it has decided it was not worth massively reducing performance and concurrency for the sake of 100% transparent failover. Even without 100% transparent failover, it is simple to guarantee *once and only once* delivery, even in the case of failure, by using a combination of duplicate detection and retrying of transactions. However this is not 100% transparent to the client code.

39.2.1.2. Handling Blocking Calls During Failover

If the client code is in a blocking call to the server, waiting for a response to continue its execution, when failover occurs, the new session will not have any knowledge of the call that was in progress. This call might otherwise hang for ever, waiting for a response that will never come.

To prevent this, HornetQ will unblock any blocking calls that were in progress at the time of failover by making them throw a **`javax.jms.JMSException`** (if using JMS), or a **`HornetQException`** with error code **`HornetQException.UNBLOCKED`**. It is up to the client code to catch this exception and retry any operations if desired.

If the method being unblocked is a call to commit(), or prepare(), then the transaction will be automatically rolled back and HornetQ will throw a **`javax.jms.TransactionRolledBackException`** (if using JMS), or a **`HornetQException`** with error code **`HornetQException.TRANSACTION_ROLLED_BACK`** if using the core API.

39.2.1.3. Handling Failover With Transactions

If the session is transactional and messages have already been sent or acknowledged in the current transaction, then the server cannot be sure that messages sent or acknowledgments have not been lost during the failover.

Consequently the transaction will be marked as rollback-only, and any subsequent attempt to commit it will throw a **`javax.jms.TransactionRolledBackException`** (if using JMS), or a **`HornetQException`** with error code **`HornetQException.TRANSACTION_ROLLED_BACK`** if using the core API.

It is up to the user to catch the exception, and perform any client side local rollback code as necessary. There is no need to manually rollback the session - it is already rolled back. The user can then just retry the transactional operations again on the same session.

HornetQ ships with a fully functioning example demonstrating how to do this, please see *Section 11.1.56, "Transaction Failover With Data Replication"*

If failover occurs when a commit call is being executed, the server, as previously described, will unblock the call to prevent a hang, since no response will come back. In this case it is not easy for the client to determine whether the transaction commit was actually processed on the live server before failure occurred.

To remedy this, the client can simply enable duplicate detection (*Chapter 37, Duplicate Message Detection*) in the transaction, and retry the transaction operations again after the call is unblocked. If the transaction had indeed been committed on the live server successfully before failover, then when the transaction is retried, duplicate detection will ensure that any durable messages resent in the transaction will be ignored on the server to prevent them getting sent more than once.

Note

By catching the rollback exceptions and retrying, catching unblocked calls and enabling duplicate detection, once and only once delivery guarantees for messages can be provided in the case of failure, guaranteeing 100% no loss or duplication of messages.

39.2.1.4. Handling Failover With Non Transactional Sessions

If the session is non transactional, messages or acknowledgments can be lost in the event of failover.

To provide *once and only once* delivery guarantees for non transacted sessions too, enabled duplicate detection, and catch unblock exceptions as described in *Section 39.2.1.2, "Handling Blocking Calls During Failover"*

39.2.2. Getting Notified of Connection Failure

JMS provides a standard mechanism for getting notified asynchronously of connection failure: **`java.jms.ExceptionListener`**. Consult the JMS javadoc for more information on how to use this.

The HornetQ core API also provides a similar feature in the form of the class **`org.hornet.core.client.SessionFailureListener`**

Any ExceptionListener or SessionFailureListener instance will always be called by HornetQ on event of connection failure, irrespective of whether the connection was successfully failed over, reconnected or reattached.

39.2.3. Application-Level Failover

In some cases automatic client failover may not be desirable, and you may prefer to handle any connection failure yourself, and code your own manual reconnection logic in your own failure handler. This defined as *application-level* failover, since the failover is handled at the user application level.

To implement application-level failover, if using JMS, set an **`ExceptionListener`** class on the JMS connection. The **`ExceptionListener`** will be called by HornetQ in the event that connection failure is detected. In **`ExceptionListener`**, close the old JMS connections, potentially look up new connection factory instances from JNDI and creating new connections. In this case you may well be using HA-JNDI to ensure that the new connection factory is looked up from a different server.

For a working example of application-level failover, please see *Section 11.1.1, "Application-Layer Failover"*.

If using the core API, the procedure is very similar: set a **`FailureListener`** on the core **`ClientSession`** instances.

Colocated and Dedicated Symmetrical Cluster Configuration

Read this chapter to configure HornetQ *live backup groups* within JBoss Enterprise Application Platform. HornetQ only supports a shared store for live backup nodes, therefore the chapter covers configuring the live backup groups in this way.

Live Backup Group

An instance of HornetQ running on JBoss Enterprise Application Platform that is configured to fail over to a specified group of HornetQ instances.

HornetQ live backup groups are configured using one of the following topologies:

Colocated
: Topology containing one live, and at least one back-up server running concurrently. Each backup node belongs to a live node on another JBoss Enterprise Application Platform instance.

Dedicated
: Topology containing one live and at least one backup server. Only one server can run at any given time.

40.1. Colocated Symmetrical Live and Backup Cluster

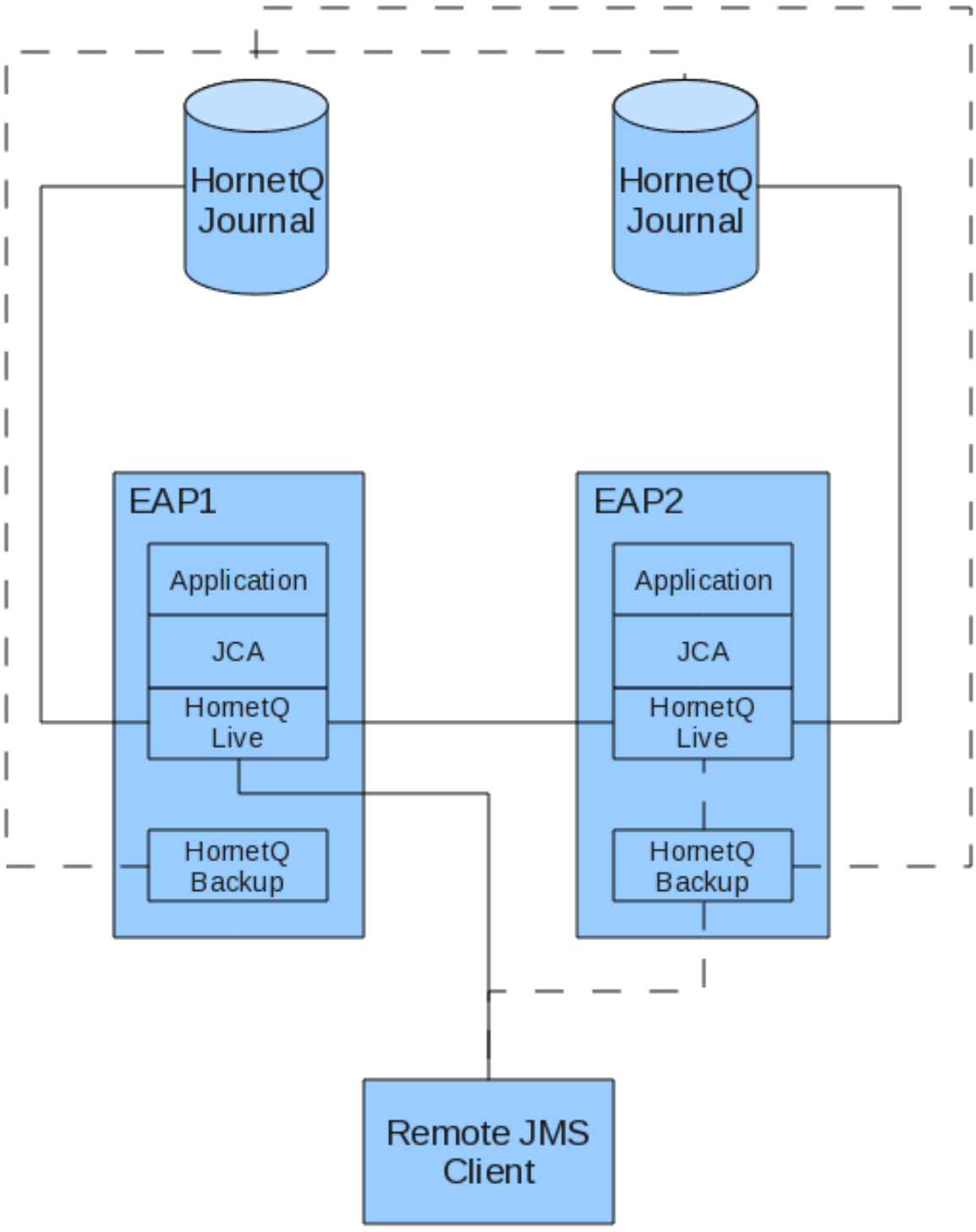
Note

JBoss Enterprise Application Platform ships with an example configuration for this topology, located in **`$JBOSS_HOME/extras/hornetq/resources/examples/symmetric-cluster-with-backups-colocated`**. The readme in this directory provides basic configuration required to run the example.

The colocated symmetrical topology contains an operational live node, and one or more backup nodes. Each backup node belongs to a live node on another JBoss Enterprise Application Platform instance.

In a simple cluster of two JBoss Enterprise Application Platform instances, each JBoss Enterprise Application Platform instance would have a live server and one backup server. as described in *Example 40.1, “Two Instance Configuration”*.

Example 40.1. Two Instance Configuration



The continuous lines in *Example 40.1, "Two Instance Configuration"*show the state of the cluster before fail over occurs. The dotted lines show the state of the cluster after fail over has occurred.

Before fail over occurs, the two live servers are connected forming a cluster. Each live server is connected to its local applications through J2EE Connector Architecture (JCA). The remote clients residing are connected to their respective live servers.

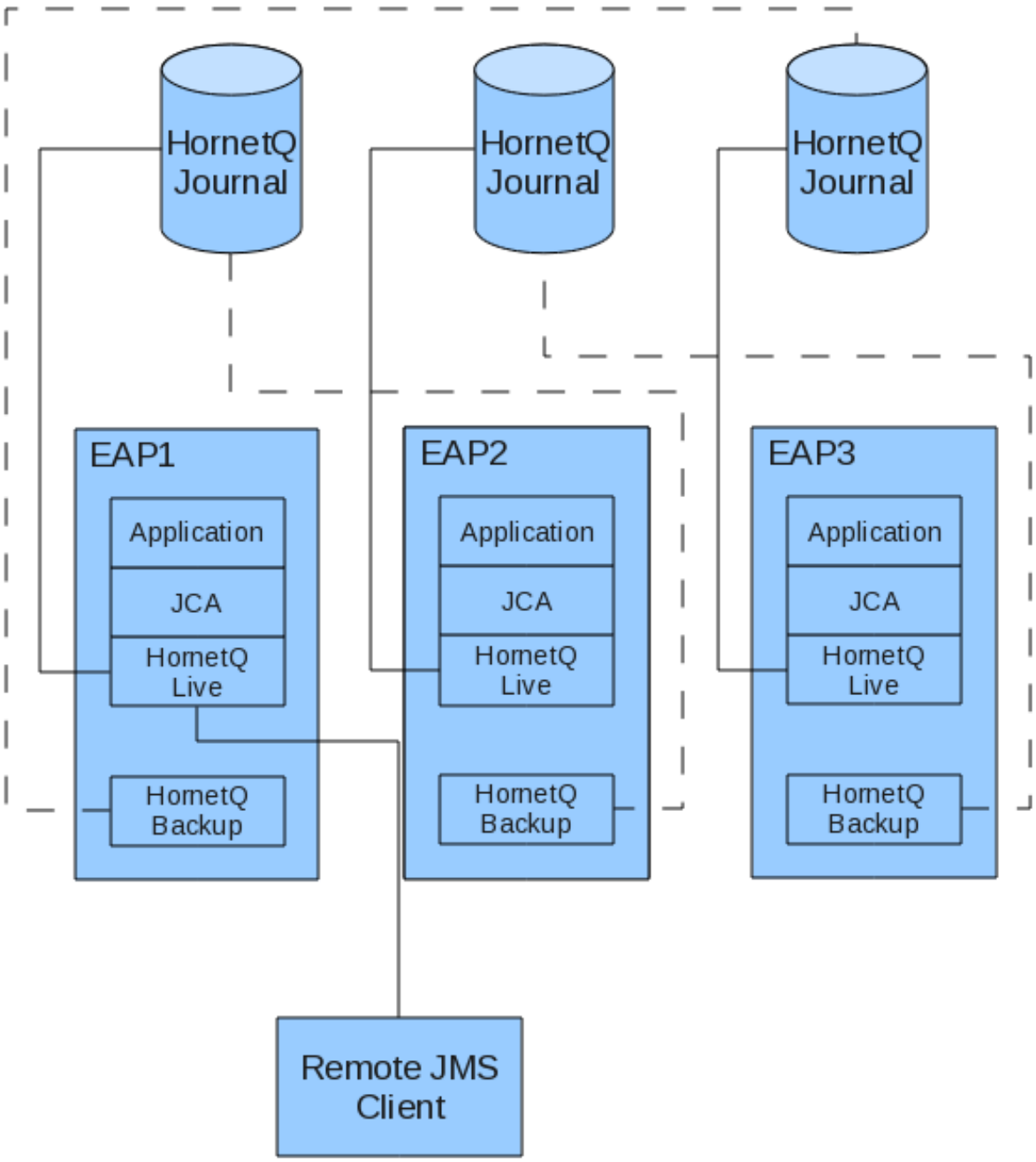
> When fail over occurs, the HornetQ Backup connects to the still available live server (which happens to be in the same virtual machine) and takes over as the live server in the cluster. Any remote clients also fail over.
>
> Depending on what consumers and producers, and Message Driven Beans (MDBs) are available on each node, messages are distributed between the nodes to satisfy Java Message Service (JMS) requirements. For example, if a producer is sending messages to a queue on a backup server with no consumers, the messages will be distributed to a live node advertising the required consumers.

Example 40.2, "Three Instance Configuration" is slightly more complex. It extends the same configuration in *Example 40.1, "Two Instance Configuration"* and adds a third live and backup HornetQ instance.

Note

The live cluster connections between each server have been removed to make the diagram clearer. All live servers form a cluster in the example.

Example 40.2. Three Instance Configuration



In this example, the instance contains three separate JBoss Enterprise Application Platform servers, with a live and backup HornetQ instance in each server.

Three node topology enables you to configure two backup instances for each server, which are shared across any of the three servers in the live backup group in case of a fail over event.

While it is possible to configure multiple live backup groups for a server, one backup per live instance is considered sufficient for most deployments.

40.1.1. Colocated Live Server

Follow the procedures to configure a Colocated Live Server instance.

Procedure 40.1. Create Live Server Profile

You must create a copy of the **`production`** profile to customize the live server configuration.

Important

Always copy an included profile rather than editing it directly for your custom profile. If you make a critical mistake during configuration, you can fall back to the base configuration at any time.

1. Navigate to **`$JBOSS_HOME/server/`**
2. Copy the **`production`** profile, and rename it to **`HornetQ_Colocated`**

Procedure 40.2. Configure Shared Store and Journaling

Follow this procedure to specify HornetQ must use a shared store for fail over, and define the location of the journal files each HornetQ instance in the live backup group uses.

1. Navigate to **`$JBOSS_HOME/server/HornetQ_Colocated/deploy/hornetq/`**
2. Open **`hornetq-configuration.xml`**
3. Add the <shared-store> element as a child of the <configuration> element.

```
<shared-store>true</shared-store>
```

4. Ensure the bindings, journal, and large messages path locations are set to a location the live backup group can access.

   You can set absolute paths as the example describes, or use the JBoss parameters that exist in the configuration file.

   If you choose the parameter option, and you do not use the default paths that these parameters resolve to, you must specify the path your bindings, journal, and large messages reside in each time you start the server.

```
<large-messages-directory>/media/shared/data/serverA/large-messages</large-messages-
directory>

<bindings-directory>/media/shared/data/serverA/bindings</bindings-directory>

<journal-directory>/media/shared/data/serverA/journal</journal-directory>

<paging-directory>/media/shared/data/ServerA/paging</paging-directory>
```

Note

Ensure you specify paths that are accessible to the live backup groups on your network.

Note

Change *`ServerA`* to the name suitable to your server instance.

By default, JMS clients will not fail over if a live server is shut down gracefully. Depending on the connection factory settings, a client will either fail, or try to reconnect to the live server.

If you need clients to fail over on a normal server shutdown, you must alter the **hornetq-configuration.xml** file according to *Procedure 40.3, "Configure JMS Client Graceful Shutdown"*.

Procedure 40.3. Configure JMS Client Graceful Shutdown

Follow this procedure to configure how JMS clients re-establish a connection if a server is shut down gracefully.

1. Navigate to **$JBOSS_HOME/server/HornetQ_Colocated/deploy/hornetq/**
2. Open **hornetq-configuration.xml**
3. Specify the <failover-on-shutdown> element in the area near the journal directory configuration in *Procedure 40.2, "Configure Shared Store and Journaling"*.

```
<failover-on-shutdown>true</failover-on-shutdown>
```

Note

You are not constrained where you put the element in the **hornetq-configuration.xml** file, however it is easier to find the less detailed settings if they are all located at the top of the file.

4. Save and close the file.

Note

If you have set <failover-on-shutdown> to false (the default setting) but still want fail over to occur, terminate the server process directly, or call **forceFailover** through the JMX Console or the Admin Console on the core server object.

The connection factories used by the client must be configured to be Highly Available. This is done by configuring connection factory attributes in the **hornetq-jms.xml**.

Procedure 40.4. Configure HA Connection Factories

1. Navigate to **$JBOSS_HOME/server/HornetQ_Colocated/deploy/hornetq/**
2. Open **hornetq-jms.xml**.
3. Add the following attributes and values as specified below.

<ha>true</ha>
: Specifies the client must support high availability, and must always be true for fail over to occur.

<retry-interval>1000</retry-interval>
: Specifies how long the client must wait (in milliseconds) before it can reconnect to the server.

<retry-interval-multiplier>1.0</retry-interval-multiplier>
Specifies the multiplier <retry-interval> uses for each subsequent reconnection pauses. By setting the value to **`1.0`**, the retry interval is the same for each client reconnection request.

<reconnect-attempts>-1</reconnect-attempts>
Specifies how many reconnect attempts a client should make before failing. Setting **`-1`** means unlimited reconnection attempts.

```
<?xml version='1.0' encoding='UTF-8'?>
<configuration xmlns="urn:hornetq" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="urn:hornetq /schema/hornetq-jms.xsd">

   <connection-factory name="NettyConnectionFactory">
      <xa>true</xa>
      <connectors>
         <connector-ref connector-name="netty"/>
      </connectors>
      <entries>
         <entry name="/ConnectionFactory"/>
         <entry name="/XAConnectionFactory"/>
      </entries>
      <ha>true</ha>

      <!-- Pause 1 second between connect attempts -->

      <retry-interval>1000</retry-interval>

      <!-- Multiply subsequent reconnect pauses by this multiplier. This can be used
       to implement an exponential back-off. For our purposes we just set to 1.0 so
       each reconnect pause is the same length -->

      <retry-interval-multiplier>1.0</retry-interval-multiplier>

      <!-- Try reconnecting an unlimited number of times (-1 means unlimited) -->

      <reconnect-attempts>-1</reconnect-attempts>
   </connection-factory>

</configuration>
```

4. Define new queues in both master and backup nodes by adding one of the following configuration blocks to the specified file.

For **`production/deploy/hornetq/hornetq-jms.xml`**

```
<queue name="testQueue">
   <entry name="/queue/testQueue"/>
   <durable>true</durable>
</queue>
```

For **`production/deploy/`*`customName`*`-hornetq-jms.xml`**

Note

Ensure the file is well-formed from an XML validation perspective by ensure the XML Namespace is present and correct in the file as specified.

```
<configuration xmlns="urn:hornetq"
               xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
               xsi:schemaLocation="urn:hornetq /schema/hornetq-jms.xsd">

   <queue name="testQueue">
      <entry name="/queue/testQueue"/>
      <durable>true</durable>
   </queue>

</configuration>
```

40.1.2. Colocated Backup Server

Read this section and the contained procedures to configure a colocated backup HornetQ server.

The backup server runs on the same JBoss Enterprise Application Platform instance as the live server configured in *Section 40.1.1, "Colocated Live Server"*. The important thing to understand is the backup server is the fail over point for a live server running on a different JBoss Enterprise Application Platform instance.

Likewise, the backup server instance does not service any Java EE components on the JBoss Enterprise Application Platform instance it is colocated on. When the live server fails over, any existing messages are redistributed within the live backup group, to service any remote clients connected to the live server at the time it became unavailable.

A backup HornetQ instance only needs a **`hornetq-jboss-beans.xml`** and a **`hornetq-configuration.xml`** configuration file. Any JMS components are created from the shared journal when the backup server becomes live (configured in *Procedure 40.2, "Configure Shared Store and Journaling"*).

Procedure 40.5. Create Backup Server

You must set up the live server first as specified in *Section 40.1.1, "Colocated Live Server"*. After you have configured the live server, continue with this procedure.

1. Navigate to **`$JBOSS_HOME/server/HornetQ_Colocated/deploy/`**
2. Create a new directory called **`hornetq-backup1`**. Move into that directory.
3. Open a text editor and create a new file called **`hornetq-jboss-beans.xml`** in the **`hornetq-backup1`** directory.
4. Copy the following configuration into **`hornetq-jboss-beans.xml`**.

```
<?xml version="1.0" encoding="UTF-8"?>
   <deployment xmlns="urn:jboss:bean-deployer:2.0">
      <!-- The core configuration -->
      <bean name="BackupConfiguration"
 class="org.hornetq.core.config.impl.FileConfiguration">
         <property name="configurationUrl">${jboss.server.home.url}/deploy/hornetq-
backup1/hornetq-configuration.xml</property>
      </bean>
      <!-- The core server -->
      <bean name="BackupHornetQServer"
 class="org.hornetq.core.server.impl.HornetQServerImpl">
         <constructor>
            <parameter>
               <inject bean="BackupConfiguration"/>
            </parameter>
            <parameter>
               <inject bean="MBeanServer"/>
```

```
            </parameter>
            <parameter>
               <inject bean="HornetQSecurityManager"/>
            </parameter>
         </constructor>
         <start ignored="true"/>
         <stop ignored="true"/>
      </bean>

      <!-- The JMS server -->
      <bean name="BackupJMSServerManager"
 class="org.hornetq.jms.server.impl.JMSServerManagerImpl">
         <constructor>
            <parameter>
               <inject bean="BackupHornetQServer"/>
            </parameter>
         </constructor>
      </bean>

</deployment>
```

5. Save and close the file.

The **hornetq-jboss-beans.xml** file in *Procedure 40.5, "Create Backup Server"* contains configuration worth exploring in more detail. The BackupConfiguration bean is configured to pick up the configuration in **hornetq-configuration.xml**. This file is created in the next procedure: *Procedure 40.6, "Create Backup Server Configuration File"*.

The HornetQ Server and JMS server beans are added after the BackupConfiguration.

Note

The names of the backup instance beans have been changed from those in the live server configuration. This is to prevent problems with beans sharing the same name. If you add more backup servers, you must rename these instances uniquely as well (for example: backup1, backup2).

The next task involves creating the backup server configuration according to *Procedure 40.6, "Create Backup Server Configuration File"*.

Procedure 40.6. Create Backup Server Configuration File

1. Navigate to **$JBOSS_HOME/server/HornetQ_Colocated/deploy/hornetq-backup1**
2. Open a text editor and create a new file called **hornetq-configuration.xml** in the **hornetq-backup1** directory.
3. Copy the following configuration into **hornetq-configuration.xml**

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration xmlns="urn:hornetq" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="urn:hornetq /schema/hornetq-configuration.xsd">

   <jmx-domain>org.hornetq.backup1</jmx-domain>

   <clustered>true</clustered>

   <backup>true</backup>

   <shared-store>true</shared-store>
```

```
<allow-failback>true</allow-failback>

<file-deployment-enabled>true</file-deployment-enabled>

<log-delegate-factory-class-name>
 org.hornetq.integration.logging.Log4jLogDelegateFactory
</log-delegate-factory-class-name>

<bindings-directory>
 /media/shared/data/hornetq-backup/bindings
</bindings-directory>

<journal-directory>/media/shared/data/hornetq-backup/journal</journal-directory>

<journal-min-files>10</journal-min-files>

<large-messages-directory>
 /media/shared/data/hornetq-backup/largemessages
</large-messages-directory>

<paging-directory>/media/shared/data/hornetq-backup/paging</paging-directory>

<connectors>
   <connector name="netty-connector">
      <factory-class>
       org.hornetq.core.remoting.impl.netty.NettyConnectorFactory
      </factory-class>
      <param key="host" value="${jboss.bind.address:localhost}"/>
      <param key="port" value="${hornetq.remoting.netty.backup.port:5446}"/>
   </connector>

   <connector name="in-vm">
      <factory-class>
       org.hornetq.core.remoting.impl.invm.InVMConnectorFactory
      </factory-class>
      <param key="server-id" value="${hornetq.server-id:0}"/>
   </connector>
</connectors>

<acceptors>
   <acceptor name="netty">
      <factory-class>
       org.hornetq.core.remoting.impl.netty.NettyAcceptorFactory
      </factory-class>
      <param key="host" value="${jboss.bind.address:localhost}"/>
      <param key="port" value="${hornetq.remoting.netty.backup.port:5446}"/>
   </acceptor>
</acceptors>

<broadcast-groups>
   <broadcast-group name="bg-group1">
      <group-address>231.7.7.7</group-address>
      <group-port>9876</group-port>
      <broadcast-period>1000</broadcast-period>
      <connector-ref>netty-connector</connector-ref>
   </broadcast-group>
</broadcast-groups>

<discovery-groups>
   <discovery-group name="dg-group1">
      <group-address>231.7.7.7</group-address>
      <group-port>9876</group-port>
      <refresh-timeout>60000</refresh-timeout>
   </discovery-group>
</discovery-groups>
```

```
   <cluster-connections>
      <cluster-connection name="my-cluster">
         <address>jms</address>
         <connector-ref>netty-connector</connector-ref>
         <discovery-group-ref discovery-group-name="dg-group1"/>
      </cluster-connection>
   </cluster-connections>

   <security-settings>
      <security-setting match="#">
         <permission type="createNonDurableQueue" roles="guest"/>
         <permission type="deleteNonDurableQueue" roles="guest"/>
         <permission type="consume" roles="guest"/>
         <permission type="send" roles="guest"/>
      </security-setting>
   </security-settings>

   <address-settings>
   <!--default for catch all-->
      <address-setting match="#">
         <dead-letter-address>jms.queue.DLQ</dead-letter-address>
         <expiry-address>jms.queue.ExpiryQueue</expiry-address>
         <redelivery-delay>0</redelivery-delay>
         <max-size-bytes>10485760</max-size-bytes>
         <message-counter-history-day-limit>10</message-counter-history-day-limit>
         <address-full-policy>BLOCK</address-full-policy>
      </address-setting>
   </address-settings>

</configuration>
```

4. Save and close the file.

The **`hornetq-configuration.xml`** file in *Procedure 40.6, "Create Backup Server Configuration File"* contains specific configuration which is discussed in *hornetq-configuration.xml Configuration Points*.

hornetq-configuration.xml Configuration Points

<jmx-domain>org.hornetq.backup1</jmx-domain>

Specifies the object name (in this case the backup server) in the Java Management Extensions (JMX) service. The default value is **`org.hornetq`**, however this name is already in use in other parts of HornetQ. You must change the name to a unique, system-wide name to avoid naming conflicts with the live server.

<clustered>true</clustered>

Specifies whether the server should join a cluster. This configuration is the same as the live server.

<backup>true</backup>

Specifies whether the server starts as a backup server, and not a live server. Specifying true sets the server to start as a backup server.

<shared-store>true</shared-store>

Specifies whether the server should reference a shared store for journaling. This configuration is the same as the live server.

<allow-failback>true</allow-failback>

Specifies whether the backup server automatically stops and returns to standby mode when the live server becomes available again. If set to **`false`**, the server must be stopped manually to trigger a return to standby mode.

<bindings-directory>, <journal-directory>, <large-messages-directory>, <paging-directory>
The paths in these elements must all resolve to the same paths the live server references. This ensures the backup server uses the same journaling files as the live server.

<connectors>
Two connectors are defined that allow clients to connect to the backup server once live: one connector for the netty connector factory (to allow client and server connections across different Virtual Machines); and one connector to allow the server to accept connections within the VM.

<acceptors>
The NettyAcceptorFactory is chosen here for VM compatibility.

<broadcast-groups>, <discovery-groups>, <cluster-connections>, <security-settings>, <address-settings>
The settings in these configuration blocks are standard settings.

Task: Create Configuration for Second Server Instance

Complete this task to configure the second server instance to cluster with the first server.

Prerequisites

- *Procedure 40.5, "Create Backup Server"*
- *Procedure 40.6, "Create Backup Server Configuration File"*

1. Navigate to ***JBOSS_HOME*** **/server/**.
2. Copy the **HornetQ_Colocated** directory, and rename it to **HornetQ_Colocated_Second**.
3. Rename ***JBOSS_HOME*****/server/HornetQ_Colocated_Second/hornetq-backup1/** to ***JBOSS_HOME*****/server/HornetQ_Colocated_Second/hornetq-backup-serverA/**
4. Open ***JBOSS_HOME*****/server/HornetQ_Colocated_Second/hornetq/hornetq-configuration.xml**
5. For all parameters with data directories specified in **hornetq-configuration.xml**, change the data paths to **/media/shared/data/hornetq-backup**.

   For example change:

   <bindings-directory> /media/shared/data/serverA/bindings </bindings-directory>

   to <bindings-directory> /media/shared/data/hornetq-backup/bindings </bindings-directory>
6. Open ***JBOSS_HOME*****/server/HornetQ_Colocated_Second/hornetq-backup-serverA/hornetq-configuration.xml**
7. For all parameters with data directories specified in **hornetq-configuration.xml**, change the data paths to **/media/shared/data/serverA**.

   For example change:

   <bindings-directory> /media/shared/data/hornetq-backup/bindings </bindings-directory>

   to <bindings-directory> /media/shared/data/serverA/bindings </bindings-directory>

40.2. Dedicated Symmetrical Live and Backup Clusters

Note

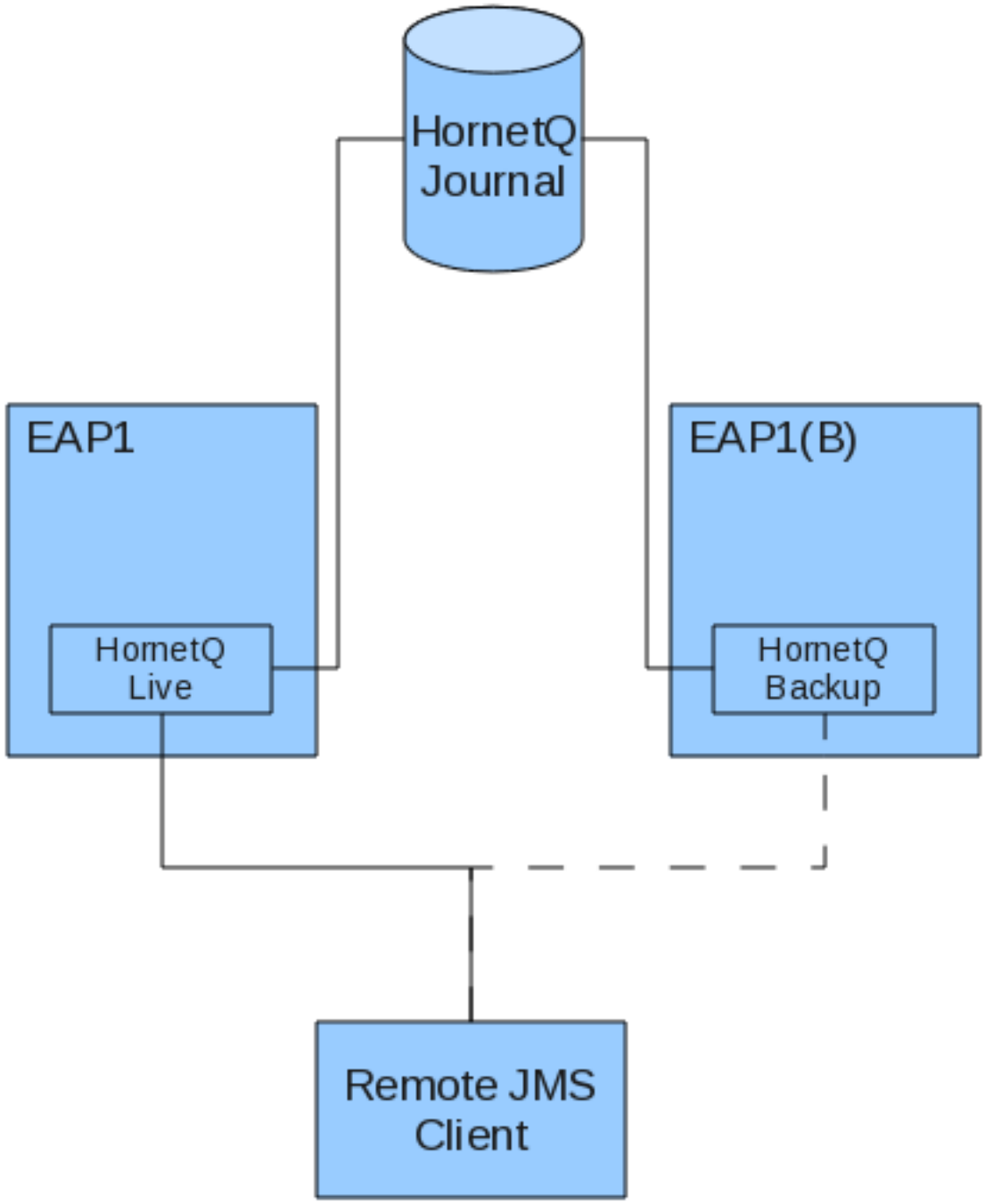
JBoss Enterprise Application Platform ships with an example configuration for this topology, located in **`$JBOSS_HOME/extras/hornetq/resources/examples/cluster-with-dedicated-backup`**.

In a dedicated symmetrical topology, the backup server resides on a separate JBoss Enterprise Application Platform instance, rather than colocated with another live server.

This means the JBoss Enterprise Application Platform instance is passive, and not used until the backup becomes live. The passive instance is therefore only useful for pure JMS applications.

The following diagram shows a possible configuration for this:

Example 40.3. Single Instance, Pure JMS, Dedicated Symmetrical Configuration



When the HornetQ live server on the EAP1 node stops responding, the HornetQ backup instance on the EAP1(B) node activates, and becomes the live server. The Remote JMS client routes all messages destined for the HornetQ live node to the HornetQ backup node.

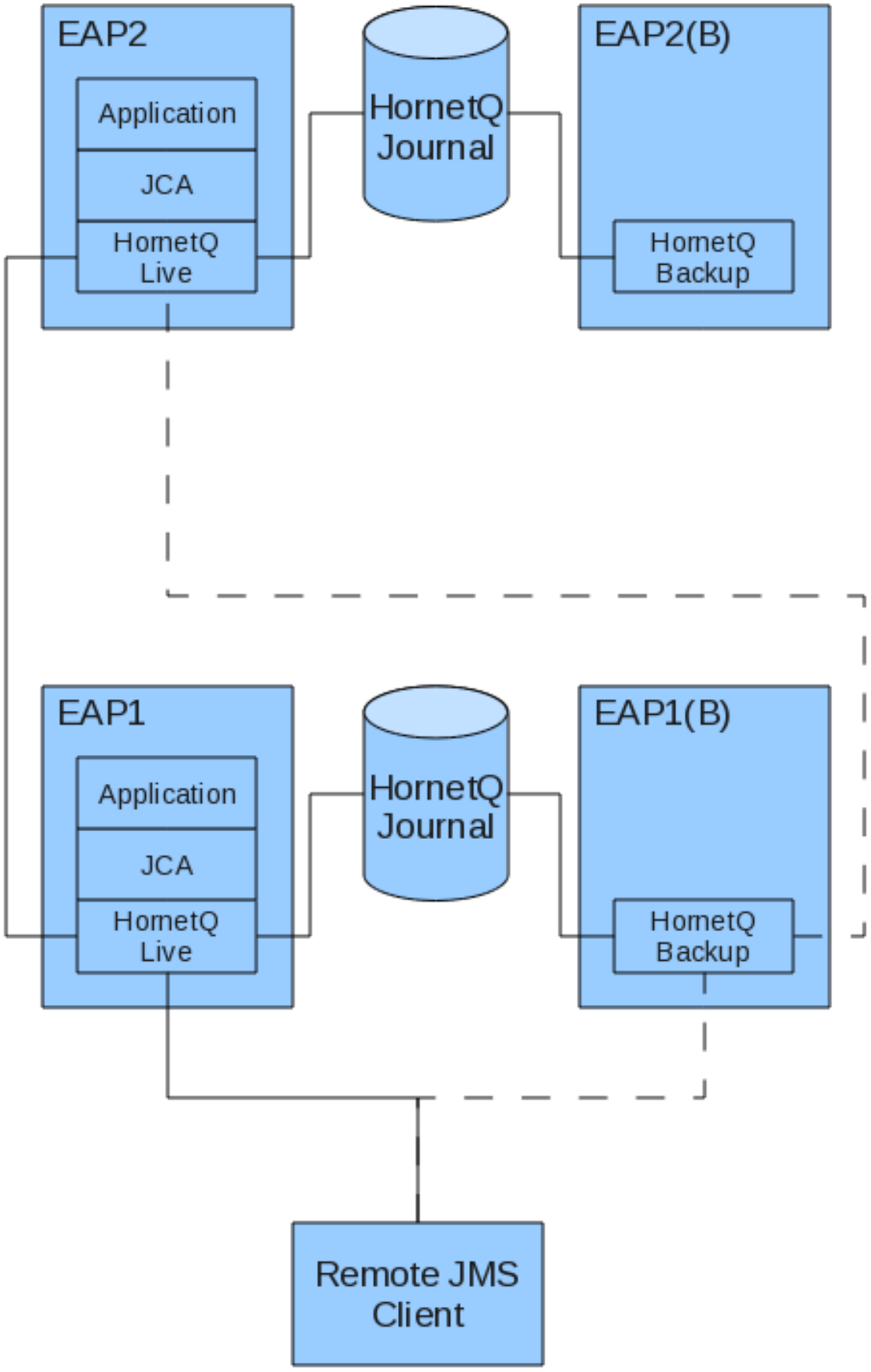
Example 40.3, "Single Instance, Pure JMS, Dedicated Symmetrical Configuration" describes how a dedicated symmetrical topology works with applications that are pure JMS, and have no JMS components (for example, Message Driven Beans).

For topologies that contain JMS components, there are two approaches you can take for dedicated symmetrical clusters containing JMS components:

1. Dedicated JCA Server, as described in *Example 40.4, “Dedicated JCA Server”*
2. Remote JCA Server, as described in *Example 40.5, “Remote JCA Server”*.

40.2.1. Dedicated JCA Live Server

Example 40.4. Dedicated JCA Server



> The EAP1(B) and EAP2(B) instances are only running backup HornetQ instances, therefore it does not make sense to host any applications on these instances. Applications are instead hosted on EAP1 and EAP2, closely located to the live HornetQ instances.
>
> When a live HornetQ server fails on EAP1, traffic fails over to the backup HornetQ servers EAP1(B). The remote JMS client reroutes messages sent from, or destined for Java EE components on the live server to the backup server using the HornetQ cluster connections.

The live server configuration is essentially the same as in *Section 40.1.1, "Colocated Live Server"*. The only difference is there is no backup in the same JBoss Enterprise Application Platform node running the live HornetQ instance. This topology also requires multiple JBoss Enterprise Application Platform instances.

Procedure 40.7. Create Dedicated Live Server Profile

You must create a copy of the **`production`** profile to customize the live server configuration.

Important

Always copy an included profile rather than editing it directly for your custom profile. If you make a critical mistake during configuration, you can fall back to the base configuration at any time.

1. Navigate to **`$JBOSS_HOME/server/`**
2. Copy the **`production`** profile, and rename it to **`HornetQ_Dedicated`**

Procedure 40.8. Configure Shared Store and Journaling

Follow this procedure to specify HornetQ must use a shared store for fail over, and define the location of the journal files each HornetQ instance in the live backup group uses.

1. Navigate to **`$JBOSS_HOME/server/HornetQ_Dedicated/deploy/hornetq/`**
2. Open **`hornetq-configuration.xml`**
3. Add the <shared-store> element as a child of the <configuration> element.

```
<shared-store>true</shared-store>
```

4. Ensure the bindings, journal, and large messages path locations are set to a location the live backup group can access.

   You can set absolute paths as the example describes, or use the JBoss parameters that exist in the configuration file.

   If you choose the parameter option, and you do not use the default paths that these parameters resolve to, you must specify the path your bindings, journal, and large messages reside in each time you start the server.

```
<large-messages-directory>/media/shared/data/large-messages</large-messages-directory>

<bindings-directory>/media/shared/data/bindings</bindings-directory>

<journal-directory>/media/shared/data/journal</journal-directory>

<paging-directory>/media/shared/data/paging</paging-directory>
```

Note

Ensure you specify paths that are accessible to the live backup groups on your network.

Procedure 40.9. Configure JMS Client Graceful Shutdown

Follow this procedure to configure how JMS clients re-establish a connection if a server is shut down gracefully.

1. Navigate to **`$JBOSS_HOME/server/HornetQ_Dedicated/deploy/hornetq/`**
2. Open **`hornetq-configuration.xml`**
3. Specify the <fail over-on-shutdown> element in the area near the journal directory configuration in *Procedure 40.2, "Configure Shared Store and Journaling"*.

```
<failover-on-shutdown>true</failover-on-shutdown>
```

Note

You are not constrained where you put the element in the **`hornetq-configuration.xml`** file, however it is easier to find the less detailed settings if they are all located at the top of the file.

4. Save and close the file.

Procedure 40.10. Configure HA Connection Factories

1. Navigate to **`$JBOSS_HOME/server/HornetQ_Dedicated/deploy/hornetq/`**
2. Open **`hornetq-jms.xml`**.
3. Add the following attributes and values as specified below.

Note

<ha>true</ha>
: Specifies the client must support high availability, and must always be true for fail over to occur.

<retry-interval>1000</retry-interval>
: Specifies how long the client must wait (in milliseconds) before it can reconnect to the server.

<retry-interval-multiplier>1.0</retry-interval-multiplier>
: Specifies the multiplier <retry-interval> uses for each subsequent reconnection pauses. By setting the value to **`1.0`**, the retry interval is the same for each client reconnection request.

<reconnect-attempts>-1</reconnect-attempts>

Specifies how many reconnect attempts a client should make before failing. Setting **-1** means unlimited reconnection attempts.

```
<?xml version='1.0' encoding='UTF-8'?>
<configuration xmlns="urn:hornetq" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="urn:hornetq /schema/hornetq-jms.xsd">

   <connection-factory name="NettyConnectionFactory">
      <xa>true</xa>
      <connectors>
         <connector-ref connector-name="netty"/>
      </connectors>
      <entries>
         <entry name="/ConnectionFactory"/>
         <entry name="/XAConnectionFactory"/>
      </entries>
      <ha>true</ha>

      <!-- Pause 1 second between connect attempts -->

      <retry-interval>1000</retry-interval>

      <!-- Multiply subsequent reconnect pauses by this multiplier. This can be used
       to implement an exponential back-off. For our purposes we just set to 1.0 so
       each reconnect pause is the same length -->

      <retry-interval-multiplier>1.0</retry-interval-multiplier>

      <!-- Try reconnecting an unlimited number of times (-1 means unlimited) -->

      <reconnect-attempts>-1</reconnect-attempts>
   </connection-factory>

</configuration>
```

4. Define new queues in both master and backup nodes by adding one of the following configuration blocks to the specified file.

   For **production/deploy/hornetq/hornetq-jms.xml**

```
<queue name="testQueue">
   <entry name="/queue/testQueue"/>
   <durable>true</durable>
</queue>
```

   For **production/deploy/*customName*-hornetq-jms.xml**

Note

Ensure the file is well-formed from an XML validation perspective by ensure the XML Namespace is present and correct in the file as specified.

```
<configuration xmlns="urn:hornetq"
               xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
               xsi:schemaLocation="urn:hornetq /schema/hornetq-jms.xsd">

   <queue name="testQueue">
```

```
      <entry name="/queue/testQueue"/>
      <durable>true</durable>
   </queue>

</configuration>
```

40.2.2. Dedicated JCA Backup Server

For the backup server, the **hornetq-configuration.xml** is unchanged. Because there is no live server, you must ensure the **hornetq-jboss-beans.xml** instantiates all the beans needed. You configure this using the same configuration as in the live server, with the exception of changing the location of the **hornetq-configuration.xml** parameter for the **Configuration** bean.

Procedure 40.11. Configure Dedicated Backup Server

Follow this procedure to configure a dedicated HornetQ backup instance, located on a separate JBoss Enterprise Application Platform server.

Prerequisites

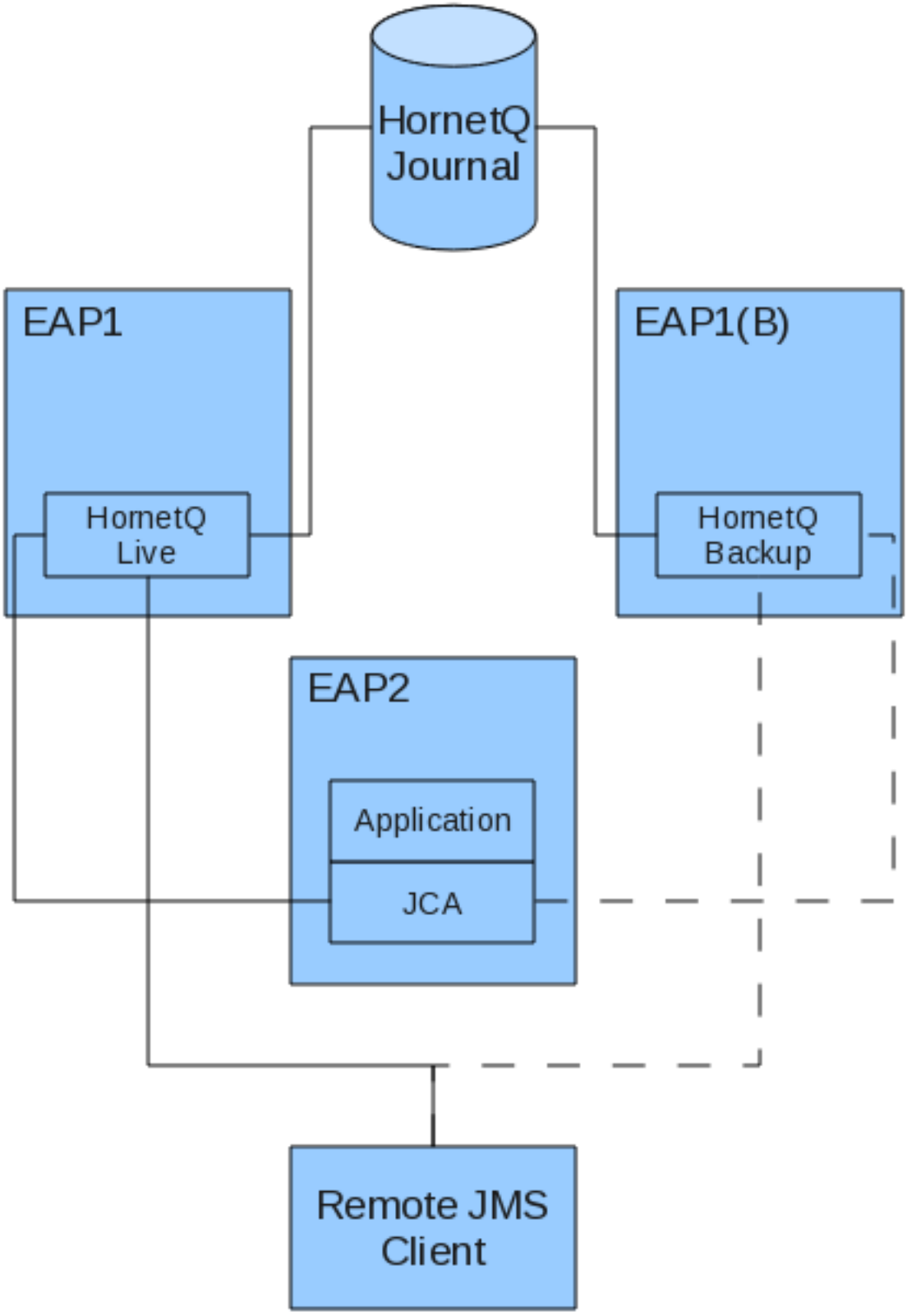
- A HornetQ Live Server, configured according to the procedures contained within *Section 40.2.1, "Dedicated JCA Live Server"*
- A correctly configured JBoss Enterprise Application Platform instance, installed on a separate server instance to the live server.
- HornetQ installed on the platform instance.

1. On the backup server, navigate using the command line to **/extras/hornetq/**.
2. Execute **sh ./switch.sh -Dbackup=true**.

   The script executes and creates a **production-backup** server profile in **$JBOSS_HOME/server/**
3. Copy the **production-backup** server profile, and rename it to **HornetQ_Dedicated_Backup**.
4. Open **$JBOSS_HOME/server/HornetQ_Dedicated_Backup/hornetq/hornetq-configuration.xml**
5. Add the <shared-store>true</shared-store> element as a child element to the <configuration> element.
6. Change the data directory locations to match the following values:
   - <large-messages-directory>/media/shared/data/large-messages</large-messages-directory>
   - <bindings-directory>/media/shared/data/bindings</bindings-directory>
   - <journal-directory>/media/shared/data/journal</journal-directory>
   - <paging-directory>/media/shared/data/paging</paging-directory>
7. Save and close all updated files.

40.2.3. Dedicated Remote Server

Example 40.5. Remote JCA Server



> In this example, the HornetQ live instance is stored on a remote server EAP1. The backup HornetQ instance is stored on EAP1(B). The JCA and the Applications are stored on a separate EAP2 instance.
>
> When fail over occurs, the Application (via JCA) is serviced by a HornetQ server in its own JBoss Enterprise Application Platform instance.

Because both HornetQ instances are located on remote servers, you must configure the JCA connection factories on the EAP2 server to correctly serve applications to the live HornetQ server and the backup HornetQ server.

Procedure 40.12. Configure JCA Connection Factories

Follow this procedure to configure the Outbound and Inbound JCA connector elements in different configuration files. Ensure you copy the Key steps are identified by a step title.

1. Copy the **`production`** server profile, and rename it to **`EAP2`**.
2. On the EAP2 instance, navigate to **`$JBOSS_HOME/server/EAP2/deploy/hornetq/jms-ds.xml`**
3. The default **`jms-ds.xml`** has the following <config-property> configuration present in the <tx-connection-factory>.

```
<?xml version="1.0" encoding="UTF-8"?>
<connection-factories>
   <!-- JMS Stuff -->

   <mbean code="org.jboss.jms.jndi.JMSProviderLoader"
    name="hornetq:service=JMSProviderLoader,name=JMSProvider">
      <attribute name="ProviderName">DefaultJMSProvider</attribute>
      <attribute name="ProviderAdapterClass">org.jboss.jms.jndi.JNDIProviderAdapter</
attribute>
      <attribute name="FactoryRef">java:/XAConnectionFactory</attribute>
      <attribute name="QueueFactoryRef">java:/XAConnectionFactory</attribute>
      <attribute name="TopicFactoryRef">java:/XAConnectionFactory</attribute>
   </mbean>

   <!-- JMS XA Resource adapter, use this to get transacted JMS in beans -->

   <tx-connection-factory>
      <jndi-name>JmsXA</jndi-name>
      <xa-transaction/>
      <rar-name>jms-ra.rar</rar-name>
      <connection-definition>org.hornetq.ra.HornetQRAConnectionFactory</connection-
definition>
      <config-property name="SessionDefaultType"
 type="java.lang.String">javax.jms.Topic</config-property>
      <config-property name="JmsProviderAdapterJNDI" type="java.lang.String">java:/
DefaultJMSProvider</config-property>
      <max-pool-size>20</max-pool-size>
      <security-domain-and-application>JmsXARealm</security-domain-and-application>
   </tx-connection-factory>
</connection-factories>
```

4. **Configure Outbound JCA Connector**
   Add extra <config-property> elements as described in the following code sample.

Important

Substitute the [live_server_IP_address] and [live_server_port_number] with the network address locations for your live server.

If you are using Discovery to set IP address/port combinations, ensure you set the appropriate parameters for <DiscoveryAddress> and <DiscoveryPort> to match your configured broadcast groups.

```
<?xml version="1.0" encoding="UTF-8"?>
<connection-factories>
   <!-- JMS Stuff -->

   <mbean code="org.jboss.jms.jndi.JMSProviderLoader"
    name="hornetq:service=JMSProviderLoader,name=JMSProvider">
      <attribute name="ProviderName">DefaultJMSProvider</attribute>
      <attribute name="ProviderAdapterClass">org.jboss.jms.jndi.JNDIProviderAdapter</
attribute>
      <attribute name="FactoryRef">java:/XAConnectionFactory</attribute>
      <attribute name="QueueFactoryRef">java:/XAConnectionFactory</attribute>
      <attribute name="TopicFactoryRef">java:/XAConnectionFactory</attribute>
   </mbean>

   <!-- JMS XA Resource adapter, use this to get transacted JMS in beans -->

   <tx-connection-factory>
      <jndi-name>JmsXA</jndi-name>
      <xa-transaction/>
      <rar-name>jms-ra.rar</rar-name>
      <connection-definition>org.hornetq.ra.HornetQRAConnectionFactory</connection-
definition>
      <config-property name="SessionDefaultType"
 type="java.lang.String">javax.jms.Topic</config-property>
      <config-property name="JmsProviderAdapterJNDI" type="java.lang.String">java:/
DefaultJMSProvider</config-property>
      <config-property name="ConnectorClassName"
 type="java.lang.String">org.hornetq.core.remoting.impl.netty.NettyConnectorFactory</
config-property>
      <config-property name="ConnectionParameters"
 type="java.lang.String">host=[live_server_IP_address];port=[live_server_port_number]</
config-property>
      <max-pool-size>20</max-pool-size>
      <security-domain-and-application>JmsXARealm</security-domain-and-application>
   </tx-connection-factory>
</connection-factories>
```

5. Open **`$JBOSS_HOME/server/EAP2/deploy/jms-ra.rar/META-INF/ra.xml`** in a text editor.

6. In **`ra.xml`**, search for <resourceadapter>.

7. **Configure the Inbound Connector**
   Replace the "The transport type" and "The transport configuration..." <config-property> elements, and their child elements with the following configuration:

Important

Substitute the [live_server_IP_address] and [live_server_port_number] with the network address locations for your live server.

If you are using Discovery to set IP address/port combinations, ensure you set the appropriate parameters for <DiscoveryAddress> and <DiscoveryPort> to match your configured broadcast groups.

If you are using Auto Discovery, ensure you comment out ConnectorClassName and ConnectionParameters directives.

```
<?xml version="1.0" encoding="UTF-8"?>

<!-- Preceeding parts of config file removed for readability -->

<resourceadapter>
   <resourceadapter-class>org.hornetq.ra.HornetQResourceAdapter</resourceadapter-class>
   <config-property>
      <description>The transport type</description>
      <config-property-name>ConnectorClassName</config-property-name>
      <config-property-type>java.lang.String</config-property-type>
      <config-property-
value>org.hornetq.core.remoting.impl.netty.NettyConnectorFactoryy</config-property-
value>
   </config-property>
   <config-property>
      <description>The transport configuration. These values must be in the form of
 key=val;key=val;</description>
      <config-property-name>ConnectionParameters</config-property-name>
      <config-property-type>java.lang.String</config-property-type>
      <config-property-
value>host=[live_server_IP_address];port=[live_server_port_number]</config-property-
value>
   </config-property>
   <config-property>
      <description>Do we support HA</description>
      <config-property-name>HA</config-property-name>
      <config-property-type>java.lang.Boolean</config-property-type>
      <config-property-value>true</config-property-value>
   </config-property>

<!-- Rest of config file removed for readability -->

<resourceadapter>
```

Libaio Native Libraries

HornetQ distributes a native library, used as a bridge between HornetQ and Linux libaio.

`libaio` is a library, developed as part of the Linux kernel project. With **`libaio`** writes are submitted to the operating system where they are processed asynchronously. Some time later the OS will call the code back when they have been processed.

This is used in the high performance journal if configured to do so. Refer to *Chapter 15, Persistence*.

These are the native libraries distributed by HornetQ:

- libHornetQAIO32.so - x86 32 bits
- libHornetQAIO64.so - x86 64 bits

When using libaio, HornetQ will always try loading these files as long as they are on the library path. Refer to *Section 6.1, "Library Path"*.

41.1. Compiling the native libraries

When using Linux on a platform other than x86_32 or x86_64 (for example Itanium 64 bits or IBM Power), you may need to compile the native library (JBoss does not distribute binaries for those platforms with the release).

41.1.1. Install requirements

Note

Presently, the native layer is only available on Linux. In a platform other than Linux, the native compilation will not work.

The native library uses autoconf, which simplifies the compilation process, however extra packages must be installed as a requirement for compilation:

- gcc - C Compiler
- gcc-c++ or g++ - Extension to gcc with support for C++
- autoconf - Tool for automating native build process
- make - Plain old make
- automake - Tool for automating make generation
- libtool - Tool for link editing native libraries
- libaio - library to disk asynchronous IO kernel functions
- libaio-dev - Compilation support for libaio
- A full JDK installed with the environment variable JAVA_HOME set to its location

To perform this installation on Red Hat Enterprise Linux or Fedora, type the following at a command line:

```
[~]$ sudo yum install automake libtool autoconf gcc-g++ gcc libaio libaio-dev make
```

41.1.2. Invoking the compilation

In the **native-src** directory of the distribution, execute the shell script **bootstrap**. This script will invoke **automake** and **make** that will create all the make files and the native library.

```
someUser@someBox:/messaging-distribution/native-src$ ./bootstrap
checking for a BSD-compatible install... /usr/bin/install -c
checking whether build environment is sane... yes
checking for a thread-safe mkdir -p... /bin/mkdir -p

...

configure: creating ./config.status
config.status: creating Makefile
config.status: creating ./src/Makefile
config.status: creating config.h
config.status: config.h is unchanged
config.status: executing depfiles commands
config.status: executing libtool commands

...
```

The library produced is located at **./native-src/src/.libs/libHornetQAIO.so**. Move the file to **bin** on the distribution or the place you have chosen on the library path. Refer to *Section 6.1, "Library Path"*.

To perform changes on the HornetQ libaio code, call **make** directly at the **native-src** directory.

Thread management

This chapter describes how HornetQ uses and pools threads and how to manage them.

First we'll discuss how threads are managed and used on the server side, then we'll look at the client side.

42.1. Server-Side Thread Management

Each HornetQ Server maintains a single thread pool for general use, and a scheduled thread pool for scheduled use. A Java scheduled thread pool cannot be configured to use a standard thread pool, otherwise we could use a single thread pool for both scheduled and non scheduled activity.

When using old (blocking) IO, a separate thread pool is also used to service connections. Since old IO requires a thread per connection, it does not make sense to get them from the standard pool as the pool will easily get exhausted if too many connections are made, resulting in the server "hanging" since it has no remaining threads to do anything else. If you require the server to handle many concurrent connections, make sure to use NIO, not old IO.

When using new IO (NIO), HornetQ will, by default, use a number of threads equal to three times the number of cores (or hyper-threads) as reported by Runtime.getRuntime().availableProcessors() for processing incoming packets. To override this value, set the number of threads by specifying the parameter **`nio-remoting-threads`** in the transport configuration. Refer to *Chapter 16, Configuring the Transport* for more information on this.

There are also a small number of other places where threads are used directly:

42.1.1. Server Scheduled Thread Pool

The server scheduled thread pool is used for most activities on the server side that require running periodically or with delays. It maps internally to a **`java.util.concurrent.ScheduledThreadPoolExecutor`** instance.

The maximum number of thread used by this pool is configured in **`hornetq-configuration.xml`** with the **`scheduled-thread-pool-max-size`** parameter. The default value is **`5`** threads. A small number of threads is usually sufficient for this pool.

42.1.2. General Purpose Server Thread Pool

This general purpose thread pool is used for most asynchronous actions on the server side. It maps internally to a **`java.util.concurrent.ThreadPoolExecutor`** instance.

The maximum number of threads used by this pool is configured in **`hornetq-configuration.xml`** with the **`thread-pool-max-size`** parameter.

If a value of **`-1`** is used this signifies that the thread pool has no upper bound and new threads will be created on demand if there are not enough threads available to satisfy a request. If activity later subsides then threads are timed-out and closed.

If a value of **n** where **n**is a positive integer greater than zero is used this signifies that the thread pool is bounded. If more requests come in and there are no free threads in the pool and the pool is full then requests will block until a thread becomes available. It is recommended that a bounded thread pool is used with caution since it can lead to dead-lock situations if the upper bound is chosen to be too low.

The default value for **`thread-pool-max-size`** is **`30`**.

See the *J2SE javadoc*[1] for more information on unbounded (cached), and bounded (fixed) thread pools.

42.1.3. Expiry Reaper Thread

A single thread is also used on the server side to scan for expired messages in queues. We cannot use either of the thread pools for this since this thread needs to run at its own configurable priority.

For more information on configuring the reaper, please see *Chapter 22, Message Expiry*.

42.1.4. Asynchronous IO

Asynchronous IO has a thread pool for receiving and dispatching events out of the native layer. You will find it on a thread dump with the prefix HornetQ-AIO-poller-pool. HornetQ uses one thread per opened file on the journal (there is usually one).

There is also a single thread used to invoke writes on libaio. We do that to avoid context switching on libaio that would cause performance issues. You will find this thread on a thread dump with the prefix HornetQ-AIO-writer-pool.

42.2. Client-Side Thread Management

On the client side, HornetQ maintains a single static scheduled thread pool and a single static general thread pool for use by all clients using the same classloader in that JVM instance.

The static scheduled thread pool has a maximum size of **5** threads, and the general purpose thread pool has an unbounded maximum size.

If required HornetQ can also be configured so that each **`ClientSessionFactory`** instance does not use these static pools but instead maintains its own scheduled and general purpose pool. Any sessions created from that **`ClientSessionFactory`** will use those pools instead.

To configure a **`ClientSessionFactory`** instance to use its own pools, simply use the appropriate setter methods immediately after creation. For example:

```
ClientSessionFactory myFactory = HornetQClient.createClientSessionFactory(...);
myFactory.setUseGlobalPools(false);
myFactory.setScheduledThreadPoolMaxSize(10);
myFactory.setThreadPoolMaxSize(-1);
```

When using the JMS API, set the same parameters on the ClientSessionFactory and use it to create the **`ConnectionFactory`** instance. For example:

```
ConnectionFactory myConnectionFactory = HornetQJMSClient.createConnectionFactory(myFactory);
```

If you're using JNDI to instantiate **`HornetQConnectionFactory`** instances, you can also set these parameters in the **`hornetq-jms.xml`** file where you describe your connection factory. For example:

```
<connection-factory name="ConnectionFactory">
   <connectors>
      <connector-ref connector-name="netty"/>
   </connectors>
```

[1] http://java.sun.com/j2se/1.5.0/docs/api/java/util/concurrent/ThreadPoolExecutor.html

```
   <entries>
      <entry name="ConnectionFactory"/>
      <entry name="XAConnectionFactory"/>
   </entries>
   <use-global-pools>false</use-global-pools>
   <scheduled-thread-pool-max-size>10</scheduled-thread-pool-max-size>
   <thread-pool-max-size>-1</thread-pool-max-size>
</connection-factory>
```

Logging

HornetQ has its own logging delegate that has no dependencies on any particular logging framework. The default delegate send all log requests to the standard JDK logging, also known as Java-Util-Logging (JUL).

By default the server picks up its JUL configuration from a **`logging.properties`** file found in the config directories. This is configured to use our own HornetQ logging formatter and will log to the console as well as a log file. For more information on configuring JUL visit the Sun website.

You can configure a different Logging Delegate programmatically or via a System Property.

To do this programmatically, do the following where **`Log4jLogDelegateFactory`** is the implementation of **`org.hornetq.spi.core.logging.LogDelegateFactory`** that you would like to use:

```
org.hornetq.core.logging.Logger.setDelegateFactory(new Log4jLogDelegateFactory())
```

To do this via a System Property simply set the property **`org.hornetq.logger-delegate-factory-class-name`** to the delegate factory being used. For example:

```
-Dorg.hornetq.logger-delegate-factory-class-
name=org.hornetq.integration.logging.Log4jLogDelegateFactory
```

In the above example, HornetQ provides some Delegate Factories for your convenience. These are:

1. **`org.hornetq.core.logging.impl.JULLogDelegateFactory`** - the default that uses JUL.
2. **`org.hornetq.integration.logging.Log4jLogDelegateFactory`** - which uses Log4J

To configure the client's logging to use the JUL delegate, ensure that a **`logging.properties`** file is provided and set the java.util.logging.config.file property on client startup.

43.1. Logging With The JBoss Application Server

When HornetQ is deployed within the JBoss Application Server version 5.x or above, it will still use JUL, however, the logging is redirected to the default JBoss logger. For more information, refer to the JBoss documentation. In earlier versions, specify the logger delegate required.

Intercepting Operations

HornetQ supports *interceptors* to intercept packets entering the server. Any supplied interceptors would be called for any packet entering the server, this allows custom code to be executed, e.g. for auditing packets, filtering or other reasons. Interceptors can change the packets they intercept.

44.1. Implementing The Interceptors

A interceptor must implement the **`Interceptor interface`**:

```
package org.hornetq.api.core.interceptor;

public interface Interceptor
{
   boolean intercept(Packet packet, RemotingConnection connection)
                throws HornetQException;
}
```

The returned boolean value is important:

- if **`true`** is returned, the process continues normally
- if **`false`** is returned, the process is aborted, no other interceptors will be called and the packet will not be handled by the server at all.

44.2. Configuring The Interceptors

The interceptors are configured in **`hornetq-configuration.xml`**:

```
<remoting-interceptors>
   <class-name>org.hornetq.jms.example.LoginInterceptor</class-name>
   <class-name>org.hornetq.jms.example.AdditionalPropertyInterceptor</class-name>
</remoting-interceptors>
```

The interceptors classes (and their dependencies) must be added to the server classpath to be properly instantiated and called.

44.3. Interceptors on the Client Side

The interceptors can also be run on the client side to intercept packets *sent by the server* by adding the interceptor to the **`ClientSessionFactory`** with the **`addInterceptor()`** method.

The interceptors classes (and their dependencies) must be added to the client classpath to be properly instantiated and called from the client side.

44.4. Example

Refer to *Section 11.1.18, “Interceptor”* for an example which shows how to use interceptors to add properties to a message on the server.

Interoperability

45.1. Stomp

Stomp[1] is a text-orientated wire protocol that allows Stomp clients to communicate with Stomp Brokers.

Stomp clients[2] are available for several languages and platforms making it a good choice for interoperability.

45.1.1. Native Stomp support

HornetQ provides native support for Stomp. To be able to send and receive Stomp messages, configure a **`NettyAcceptor`** with a **`protocol`** parameter set to **`stomp`**:

```
<acceptor name="stomp-acceptor">
   <factory-class>org.hornetq.core.remoting.impl.netty.NettyAcceptorFactory</factory-class>
   <param key="protocol"  value="stomp"/>
   <param key="port"  value="61613"/>
</acceptor>
```

With this configuration, HornetQ will accept Stomp connections on the port **`61613`** (which is the default port of the Stomp brokers).

See the **`stomp`** example which shows how to configure a HornetQ server with Stomp.

45.1.1.1. Limitations

Message acknowledgments are not transactional. The ACK frame can not be part of a transaction (it will be ignored if the **`transaction`** header is set).

45.1.2. Mapping Stomp destinations to HornetQ addresses and queues

Stomp clients deals with *destinations* when sending messages and subscribing. Destination names are simply strings which are mapped to some form of destination on the server - how the server translates these is left to the server implementation.

In HornetQ, these destinations are mapped to *addresses* and *queues*. When a Stomp client sends a message (using a **`SEND`** frame), the specified destination is mapped to an address. When a Stomp client subscribes (or unsubscribes) for a destination (using a **`SUBSCRIBE`** or **`UNSUBSCRIBE`** frame), the destination is mapped to a HornetQ queue.

45.1.3. STOMP and connection-ttl

Well behaved STOMP clients will always send a DISCONNECT frame before closing their connections. In this case the server will clear up any server side resources such as sessions and consumers synchronously. However if STOMP clients exit without sending a DISCONNECT frame or if they crash the server will have no way of knowing immediately whether the client is still alive

[1] http://stomp.codehaus.org/
[2] http://stomp.codehaus.org/Clients

or not. STOMP connections therefore default to a connection-ttl value of 1 minute (see chapter on *connection-ttl* for more information. This value can be overridden using connection-ttl-override.

Note

Please note that the STOMP protocol does not contain any heartbeat frame. It is therefore the user's responsibility to make sure data is sent within connection-ttl or the server will assume the client is dead and clean up server side resources.

45.1.4. Stomp and JMS interoperabilty

45.1.4.1. Using JMS destinations

As explained in *Chapter 9, Mapping JMS Concepts to the Core API*, JMS destinations are also mapped to HornetQ addresses and queues. If Stomp is to be used to send messages to JMS destinations, the Stomp destinations must follow the same convention:

- send or subscribe to a JMS *Queue* by prepending the queue name by **`jms.queue.`**.

  For example, to send a message to the **`orders`** JMS Queue, the Stomp client must send the frame:

```
SEND
destination:jms.queue.orders

hello queue orders
^@
```

- send or subscribe to a JMS *Topic* by prepending the topic name by **`jms.topic.`**.

  For example to subscribe to the **`stocks`** JMS Topic, the Stomp client must send the frame:

```
SUBSCRIBE
destination:jms.topic.stocks

^@
```

45.1.4.2. Send and Consuming Stomp message from JMS or HornetQ Core API

Stomp is mainly a text-orientated protocol. To make it simpler to interoperate with JMS and HornetQ Core API, our Stomp implementation checks for presence of the **`content-length`** header to decide how to map a Stomp message to a JMS Message or a Core message.

If the Stomp message does *not* have a **`content-length`** header, it will be mapped to a JMS *TextMessage* or a Core message with a *single nullable SimpleString in the body buffer*.

Alternatively, if the Stomp message *has* a **`content-length`** header, it will be mapped to a JMS *BytesMessage* or a Core message with a *byte[] in the body buffer*.

The same logic applies when mapping a JMS message or a Core message to Stomp. A Stomp client can check the presence of the **`content-length`** header to determine the type of the message body (UTF-8 String or bytes).

45.1.5. Stomp Over Web Sockets

HornetQ also support Stomp over *Web Sockets*[3]. Modern web browser which support Web Sockets can send and receive Stomp messages from HornetQ.

To enable Stomp over Web Sockets, configure a **`NettyAcceptor`** with a **`protocol`** parameter set to **`stomp_ws`**:

```
<acceptor name="stomp-ws-acceptor">
   <factory-class>org.hornetq.core.remoting.impl.netty.NettyAcceptorFactory
  </factory-class>
   <param key="protocol" value="stomp_ws"/>
   <param key="port" value="61614"/>
</acceptor>
```

With this configuration, HornetQ will accept Stomp connections over Web Sockets on the port **`61614`** with the URL path **`/stomp`**. Web browser can then connect to **`ws://<server>:61614/stomp`** using a Web Socket to send and receive Stomp messages.

A companion JavaScript library to ease client-side development is available from *GitHub*[4] (please see its *documentation*[5] for a complete description).

The **`stomp-websockets`** example shows how to configure HornetQ server to have web browsers and Java applications exchanges messages on a JMS topic.

45.1.6. StompConnect

StompConnect[6] is a server that can act as a Stomp broker and proxy the Stomp protocol to the standard JMS API. Consequently, using StompConnect it is possible to turn HornetQ into a Stomp Broker and use any of the available stomp clients. These include clients written in C, C++, c# and .net etc.

To run StompConnect first start the HornetQ server and make sure that it is using JNDI.

Stomp requires the file **`jndi.properties`** to be available on the classpath. This should look something like:

```
java.naming.factory.initial=org.jnp.interfaces.NamingContextFactory
java.naming.provider.url=jnp://localhost:1099
java.naming.factory.url.pkgs=org.jboss.naming:org.jnp.interfaces
```

Make sure this file is in the classpath along with the StompConnect jar and the HornetQ jars and simply run **`java org.codehaus.stomp.jms.Main`**.

[3] http://dev.w3.org/html5/websockets/
[4] http://github.com/jmesnil/stomp-websocket
[5] http://jmesnil.net/stomp-websocket/doc/
[6] http://stomp.codehaus.org/StompConnect

Performance Tuning

In this chapter we'll discuss how to tune HornetQ for optimum performance.

46.1. Tuning persistence

- Put the message journal on its own physical volume. If the disk is shared with other processes e.g. transaction coordinator, database or other journals which are also reading and writing from the disk, this may greatly reduce performance since the disk head may be skipping between the different files. One advantage of an append only journal is that disk head movement is minimized - this advantage is destroyed if the disk is shared. If using paging or large messages, ideally, ensure that they are put on separate volumes.
- Minimum number of journal files. Set **`journal-min-files`** to a number of files that would fit the average sustainable rate. If new files being created on the journal data directory too often, i.e. lots of data is being persisted, increase the minimal number of files, this way the journal would reuse more files instead of creating new data files.
- Journal file size. The journal file size should be aligned to the capacity of a cylinder on the disk. The default value 10MiB should be enough on most systems.
- Use AIO journal. If using Linux, try to keep the journal type as AIO. AIO will scale better than Java NIO.
- Tune **`journal-buffer-timeout`**. The timeout can be increased to increase throughput at the expense of latency.
- If running AIO, it may be possible to increase performance by increasing **`journal-max-io`**. DO NOT change this parameter if running NIO.

46.2. Tuning JMS

There are a few areas where some tweaks can be done if using the JMS API

- Disable message id. Use the **`setDisableMessageID()`** method on the **`MessageProducer`** class to disable message ids if not needed. This decreases the size of the message and also avoids the overhead of creating a unique ID.
- Disable message timestamp. Use the **`setDisableMessageTimeStamp()`** method on the **`MessageProducer`** class to disable message timestamps not required.
- Avoid **`ObjectMessage`**. **`ObjectMessage`** is convenient but it comes at a cost. The body of a **`ObjectMessage`** uses Java serialization to serialize it to bytes. The Java serialized form of even small objects is very verbose, using a large volume of space on the wire, also Java serialization is slow in comparison to custom marshalling techniques. Only use **`ObjectMessage`** if one of the other message types are unsuitable, i.e. if the type of payload is unknown until run-time.
- Avoid **`AUTO_ACKNOWLEDGE`**. **`AUTO_ACKNOWLEDGE`** mode requires an acknowledgment to be sent from the server for each message received on the client, this means more traffic on the network. If possible, use **`DUPS_OK_ACKNOWLEDGE`**, or use **`CLIENT_ACKNOWLEDGE`** or a transacted session and batch up many acknowledgments with one acknowledge/commit.
- Avoid durable messages. By default JMS messages are durable. If really durable messages are not required, set the messages to be non-durable. Durable messages incur much more overhead in persisting them to storage.

- Batch many sends or acknowledgments in a single transaction. HornetQ will only require a network round trip on the commit, not on every send or acknowledgment.

46.3. Other Tunings

There are various other places in HornetQ where tuning can be performed:

- Use Asynchronous Send Acknowledgments. To send durable messages non transactionally and a guarantee is required that they have reached the server by the time the call to send() returns, do not set durable messages to be sent blocking, rather use asynchronous send acknowledgments to get the acknowledgments of send back in a separate stream. Refer to *Chapter 20, Guarantees of sends and commits* for more information on this.
- Use pre-acknowledge mode. With pre-acknowledge mode, messages are acknowledged **`before`** being sent to the client. This reduces the amount of acknowledgment traffic on the wire. For more information on this, refer to *Chapter 29, Pre-Acknowledge Mode*.
- Disable security. A small performance boost is possible by disabling security by setting the **`security-enabled`** parameter to **`false`** in **`hornetq-configuration.xml`**.
- Disable persistence. If message persistence is not required, turn it off altogether by setting **`persistence-enabled`** to false in **`hornetq-configuration.xml`**.
- Sync transactions lazily. Setting **`journal-sync-transactional`** to **`false`** in **`hornetq-configuration.xml`** gives better transactional persistent performance at the expense of some possibility of loss of transactions on failure. Refer to *Chapter 20, Guarantees of sends and commits* for more information.
- Sync non transactional lazily. Setting **`journal-sync-non-transactional`** to **`false`** in **`hornetq-configuration.xml`** can provide better non-transactional persistent performance at the expense of some possibility of loss of durable messages on failure. Refer to *Chapter 20, Guarantees of sends and commits* for more information.
- Send messages non blocking. Setting **`block-on-durable-send`** and **`block-on-non-durable-send`** to **`false`** in **`hornetq-jms.xml`** (if using JMS and JNDI) or directly on the ClientSessionFactory. It is therefore not required to wait an entire network round trip for every message sent. Refer to *Chapter 20, Guarantees of sends and commits* for more information.
- For very fast consumers, increase consumer-window-size. This effectively disables consumer flow control.
- Socket NIO vs Socket Old IO. By default HornetQ uses old (blocking) on the server and the client side (Refer to *Chapter 16, Configuring the Transport* for more information). NIO is much more scalable but can give some latency hit compared to old blocking IO. To be able to service many thousands of connections on the server, then ensure the use of NIO on the server. However, for fewer connections on the server, retaining the old IO for the server acceptors may gain a small performance advantage.
- Use the core API not JMS. Using the JMS API will have slightly lower performance than using the core API, since all JMS operations need to be translated into core operations before the server can handle them. If using the core API, try to use methods that take **`SimpleString`** as much as possible. **`SimpleString`**, unlike **`java.lang.String`** does not require copying before it is written to the wire, so if you re-use **`SimpleString`** instances between calls, some unnecessary copying can be avoided.

46.4. Tuning Transport Settings

- TCP buffer sizes. Fast networks and fast machines may get a performance boost by increasing the TCP send and receive buffer sizes. Refer to *Chapter 16, Configuring the Transport* for more information.

Note

Note that some operating systems like later versions of Linux include TCP auto-tuning and setting TCP buffer sizes manually can prevent auto-tune from working and actually give you worse performance!

- Increase limit on file handles on the server. If a lot of concurrent connections on the servers is expected, or if clients are rapidly opening and closing connections, ensure that the user running the server has permission to create sufficient file handles.

  This varies from operating system to operating system. On Linux systems, increase the number of allowable open file handles in the file **`/etc/security/limits.conf`**, e.g. add the lines

```
serveruser     soft    nofile  20000
serveruser     hard    nofile  20000
```

  This would allow up to 20 000 file handles to be open by the user **`serveruser`**.

- Use **`batch-delay`** and set **`direct-deliver`** to false for the best throughput for very small messages. HornetQ comes with a preconfigured connector/acceptor pair (**`netty-throughput`**) in **`hornetq-configuration.xml`** and JMS connection factory (**`ThroughputConnectionFactory`**) in **`hornetq-jms.xml`**which can be used to give the very best throughput, especially for small messages. Refer to *Chapter 16, Configuring the Transport* for more information.

46.5. Avoiding Anti-Patterns

- Re-use connections / sessions / consumers / producers. Probably the most common messaging anti-pattern observed, is users who create a new connection/session/producer for every message sent, or every message consumed. This is a poor use of resources. Always reuse these objects as they take time to create and may involve several network round trips.

Note

Some popular libraries such as the Spring JMS Template are known to use these anti-patterns. If using Spring JMS Template, it is possibly a cause for poor performance and not HornetQ. The Spring JMS Template can only safely be used in an app server which caches JMS sessions (e.g. using JCA), and only then for sending messages. It cannot be safely be used for synchronously consuming messages, even in an app server.

- Avoid fat messages. Verbose formats such as XML take up a lot of space on the wire and performance will suffer as result. Avoid XML in message bodies if possible.

- Do not create temporary queues for each request. This common anti-pattern involves the temporary queue request-response pattern. With the temporary queue request-response pattern a message is sent to a target and a reply-to header is set with the address of a local temporary queue. When the recipient receives the message they process it then send back a response to the address specified in the reply-to. A common mistake made with this pattern is to create a new temporary queue on each message sent. This will drastically reduce performance. Instead the temporary queue should be re-used for many requests.
- Do not use Message-Driven Beans unnecessarily. MDB usage greatly increases the code path for each message received compared to a straightforward message consumer, as a lot of extra application server code is executed. Before using MDBs, investigate the use of a normal message consumer to complete the task.

Configuration Reference

This section is a quick index for looking up configuration values. Click on the element name to go to the specific chapter.

47.1. Server Configuration

47.1.1. hornetq-configuration.xml

This is the main core server configuration file.

Table 47.1. Server Configuration

Element Name	Type	Description	Default
backup	Boolean	if **`true`**, this server is a backup to another node in the cluster.	false
allow-failback	Boolean	Specifies whether the backup server automatically stops and returns to standby mode when the live server becomes available again. If set to **`false`**, the server must be stopped manually to trigger a return to standby mode.	true
failover-on-shutdown	Boolean	Specifies how failover behaves when a live server is shutdown correctly. If set to **`true`**, the backup HornetQ instance takes over when the live server is shut down gracefully.	false
shared-store		Specifies whether the server should reference a shared store for journaling.	false
grouping-handler	Parent element	Used to specify the LOCAL and REMOTE grouping handlers. Refer to *Section 11.1.6, "Clustered Grouping"* for detailed information about this element.	
remoting-interceptors	Parent element	Used to specify class names, using <class-name>. Refer to *Chapter 44, Intercepting Operations* for detailed information about this element.	
address-full-policy	String	Supports three values: PAGE, DROP, and BLOCK. Refer to *Section 24.3, "Paging Mode"*	PAGE
send-to-dla-on-no-route	Boolean	Specifies how messages are handled when a server does not routes the message to a queue. If set to **`true`**, the	false

Element Name	Type	Description	Default
		message is sent to the dead letter address (DLA) for the routed address, if a DLA exists. If a DLA does not exist, the message is dropped as a last resort. Refer to *Section 25.3, "Configuring Queues Through Address Settings"* for more information about Address Setting elements.	
backup-connector-ref	String	the name of the remoting connector to connect to the backup node.	
bindings-directory	String	the directory in which to store the persisted bindings.	data/bindings
clustered	Boolean	if **`true`**, the server is clustered.	false
connection-ttl-override	Long	if set, overrides the time (in ms) to keep a connection alive without receiving a ping.	-1
create-bindings-dir	Boolean	true means that the server will create the bindings directory on start up.	true
create-journal-dir	Boolean	if **`true`**, the **`journal`** directory will be created.	true
file-deployment-enabled	Boolean	if **`true`**, the server loads configuration from the configuration files.	true
id-cache-size	Integer	the size of the cache for pre-creating message IDs.	2000
journal-buffer-size	Long	The size of the internal buffer on the journal.	128 kilobytes
journal-buffer-timeout	Long	the timeout (in nanoseconds) used to flush internal buffers on the journal.	20000
journal-compact-min-files	Integer	the minimum number of data files before compacting occurs.	10
journal-compact-percentage	Integer	the percentage of live data on which we consider compacting the journal.	30
journal-directory	String	the directory to store the journal files in.	data/journal
journal-file-size	Long	the size (in bytes) of each journal file.	128 * 1024
journal-max-io	Integer	the maximum number of write requests that can be in the AIO queue at any one time.	500

Element Name	Type	Description	Default
journal-min-files	Integer	the number of journal files to pre-create.	2
journal-sync-transactional	Boolean	if **`true`**, wait for transaction data to be synchronized to the journal before returning response to client.	true
journal-sync-non-transactional	Boolean	if **`true`**, wait for non-transaction data to be synced to the journal before returning response to client.	true
journal-type	String	the type of journal to use.	ASYNCIO
jmx-management-enabled	Boolean	if **`true`**, the management API is available via JMX.	true
jmx-domain	String	the JMX domain used to registered HornetQ MBeans in the MBeanServer.	org.hornetq
large-messages-directory	String	the directory in which to store large messages. The default location is **`data/largemessages`**	
management-address	String	the name of the management address to send management messages to. The default value is **`jms.queue.hornetq.management`**	
cluster-user	String	the user used by cluster connections to communicate between the clustered nodes. The default value is **`HORNETQ.CLUSTER.ADMIN.USER`**	See Description
cluster-password	String	the password used by cluster connections to communicate between the clustered nodes.	CHANGE ME!!
management-notification-address	String	the name of the address that consumers bind to receive management notifications. The default value is **`hornetq.notifications`**	See Description
message-counter-enabled	Boolean	if **`true`**, message counters are enabled.	false
message-counter-max-day-history	Integer	how many days to keep message counter history.	10
message-counter-sample-period	Long	the sample period (in ms) to use for message counters.	10000
message-expiry-scan-period	Long	how often (in ms) to scan for expired messages.	30000

Element Name	Type	Description	Default
message-expiry-thread-priority	Integer	the priority of the thread expiring messages.	3
paging-directory	String	the directory to store paged messages in.	data/paging
persist-delivery-count-before-delivery	Boolean	if **`true`**, delivery count is persisted before delivery; if **`false`**, this occurs only after a message has been canceled.	false
persistence-enabled	Boolean	true means that the server will use the file based journal for persistence.	true
persist-id-cache	Boolean	true means that id's are persisted to the journal.	true
scheduled-thread-pool-max-size	Integer	the number of threads that the main scheduled thread pool has.	5
security-enabled	Boolean	if **`true`**, security is enabled.	true
security-invalidation-interval	Long	how long (in ms) to wait before invalidating the security cache.	10000
thread-pool-max-size	Integer	the number of threads that the main thread pool has; **`-1`** sets unlimited threads.	30
async-connection-execution-enabled	Boolean	should incoming packets on the server be handed off to a thread from the thread pool for processing or should they be handled on the remoting thread?	true
transaction-timeout	Long	how long (in ms) before a transaction can be removed from the resource manager after create time.	60000
transaction-timeout-scan-period	Long	how often (in ms) to scan for timeout transactions.	1000
wild-card-routing-enabled	Boolean	true means that the server supports wild card routing.	true
memory-measure-interval	Long	frequency to sample JVM memory in ms (or -1 to disable memory sampling).	-1
memory-warning-threshold	Integer	Percentage of available memory which threshold a warning log.	25
connectors	String	a list of remoting connectors configurations to create.	
connector.name (attribute)	String	Name of the connector - mandatory.	

Element Name	Type	Description	Default
connector.factory-class	String	Name of the ConnectorFactory implementation - mandatory.	
connector.param	String	A key-value pair used to configure the connector. A connector can have many param.	
connector.param.key (attribute)	String	Key of a configuration parameter - mandatory.	
connector.param.value (attribute)	String	Value of a configuration parameter - mandatory.	
acceptors	String	a list of remoting acceptors to create.	
acceptor.name (attribute)	String	Name of the acceptor - optional.	
acceptor.factory-class	String	Name of the AcceptorFactory implementation - mandatory.	
acceptor.param	String	A key-value pair used to configure the acceptor. An acceptor can have many param.	
acceptor.param.key (attribute)	String	Key of a configuration parameter - mandatory.	
acceptor.param.value (attribute)	String	Value of a configuration parameter - mandatory.	
broadcast-groups		a list of broadcast groups to create.	
broadcast-group.name (attribute)	String	a unique name for the broadcast group - mandatory.	
broadcast-group.local-bind-address	String	local bind address that the datagram socket is bound to. The default value is the wildcard IP address chosen by the kernel	See Description
broadcast-group.local-bind-port	Integer	local port to which the datagram socket is bound to.	-1 (anonymous port)
broadcast-group.group-address	String	multicast address to which the data will be broadcast - mandatory.	
broadcast-group.group-port	Integer	UDP port number used for broadcasting - mandatory.	
broadcast-group.broadcast-period	Long	period in milliseconds between consecutive broadcasts.	2000 (ms)
broadcast-group.connector-ref	Integer	A pair connector and optional backup connector that will be broadcasted. A broadcast-	

Element Name	Type	Description	Default
		group can have multiple connector-ref.	
broadcast-group.connector-ref.connector-name (attribute)	String	Name of the live connector - mandatory.	
broadcast-group.connector-ref.backup-connector-name (attribute)	String	Name of the backup connector - optional.	
discovery-groups	String	a list of discovery groups to create.	
discovery-group.name (attribute)	String	A unique name for the discovery group - mandatory.	
discovery-group.local-bind-address	String	The discovery group will be bound only to this local address.	
discovery-group.group-address	String	Multicast IP address of the group to listen on - mandatory.	
discovery-group.group-port	Integer	UDP port of the multicast group - mandatory	
discovery-group.refresh-timeout	Integer	Period the discovery group waits after receiving the last broadcast from a particular server before removing that servers connector pair entry from its list.	5000 (ms)
diverts	String	A list of diverts to use.	
divert.name (attribute)	String	A unique name for the divert - mandatory.	
divert.routing-name	String	The routing name for the divert - mandatory.	
divert.address	String	The address this divert will divert from - mandatory.	
divert.forwarding-address	String	The forwarding address for the divert - mandatory.	
divert.exclusive	Boolean	Is this divert exclusive?	false
divert.filter	String	An optional core filter expression.	null
divert.transformer-class-name	String	An optional class name of a transformer.	
queues	String	A list of pre configured queues to create.	
queues.name (attribute)	String	Unique name of this queue.	
queues.address	String	Address for this queue - mandatory.	
queues.filter	String	Optional core filter expression for this queue.	null

Element Name	Type	Description	Default
queues.durable	Boolean	Is this queue durable?	true
bridges	String	A list of bridges to create.	
bridges.name (attribute)	String	Unique name for this bridge.	
bridges.queue-name	String	Name of queue that this bridge consumes from - mandatory.	
bridges.forwarding-address	String	Address to forward to. If omitted original address is used.	null
bridges.filter	String	Optional core filter expression.	null
bridges.transformer-class-name	String	Optional name of transformer class.	null
bridges.retry-interval	Long	Period (in ms) between successive retries.	2000 (ms)
bridges.retry-interval-multiplier	Double	Multiplier to apply to successive retry intervals.	1.0
bridges.reconnect-attempts	Integer	Maximum number of retry attempts, -1 signifies infinite.	-1
bridges.failover-on-server-shutdown	Boolean	Should failover be prompted if target server is cleanly shutdown?	false
bridges.use-duplicate-detection	Boolean	Should duplicate detection headers be inserted in forwarded messages?	true
bridges.discovery-group-ref	String	Name of discovery group used by this bridge.	null
bridges.connector-ref.connector-name (attribute)	String	Name of connector to use for live connection.	
bridges.connector-ref.backup-connector-name (attribute)	String	Optional name of connector to use for backup connection.	null
cluster-connections	String	A list of cluster connections.	
cluster-connections.name (attribute)	String	Unique name for this cluster connection.	
cluster-connections.address	String	Name of address this cluster connection applies to.	
cluster-connections.forward-when-no-consumers	Boolean	Should messages be load balanced if there are no matching consumers on target?	false
cluster-connections.max-hops	Integer	Maximum number of hops cluster topology is propagated.	1
cluster-connections.retry-interval	Long	Period (in ms) between successive retries.	2000
cluster-connections.use-duplicate-detection	Boolean	Should duplicate detection headers be inserted in forwarded messages?	true

Element Name	Type	Description	Default
cluster-connections.discovery-group-ref	String	Name of discovery group used by this bridge.	null
cluster-connections.connector-ref.connector-name (attribute)	String	Name of connector to use for live connection.	
cluster-connections.connector-ref.backup-connector-name (attribute)	String	Optional name of connector to use for backup connection.	null
security-settings	String	A list of security settings.	
security-settings.match (attribute)	String	The string to use for matching security against an address.	
security-settings.permission	String	A permission to add to the address.	
security-settings.permission.type (attribute)	String	The type of permission.	
security-settings.permission.roles (attribute)	String	A comma-separated list of roles to apply the permission to.	
address-settings	String	A list of address settings.	
address-settings.dead-letter-address	String	The address to send dead messages to.	
address-settings.max-delivery-attempts	Integer	How many times to attempt to deliver a message before sending to dead letter address.	10
address-settings.expiry-address	String	The address to send expired messages to.	
address-settings.redelivery-delay	Long	The time (in ms) to wait before redelivering a canceled message.	0
address-settings.last-value-queue	boolean	Whether to treat the queue as a last value queue.	false
address-settings.page-size-bytes	Long	The page size (in bytes) to use for an address.	10 * 1024 * 1024
address-settings.max-size-bytes	Long	The maximum size (in bytes) to use in paging for an address.	-1
address-settings.redistribution-delay	Long	How long (in ms) to wait after the last consumer is closed on a queue before redistributing messages.	-1

47.1.2. hornetq-jms.xml

This is the configuration file used by the server side JMS service to load JMS Queues, Topics and Connection Factories.

Table 47.2. JMS Server Configuration

Element Name	Type	Description	Default
connection-factory		A list of connection factories to create and add to JNDI.	
connection-factory.auto-group	Boolean	Whether or not message grouping is automatically used.	false
connection-factory.connectors	String	A list of connectors used by the connection factory.	
connection-factory.connectors.connector-ref.connector-name (attribute)	String	Name of the connector to connect to the live server.	
connection-factory.connectors.connector-ref. backup-connector-name (attribute)	String	Name of the connector to connect to the backup server.	
connection-factory.discovery-group-ref.discovery-group-name (attribute)	String	Name of discovery group used by this connection factory.	
connection-factory.discovery-initial-wait-timeout	Long	The initial time to wait (in ms) for discovery groups to wait for broadcasts.	10000
connection-factory.block-on-acknowledge	Boolean	Whether or not messages are acknowledged synchronously.	false
connection-factory.block-on-non-durable-send	Boolean	Whether or not non-durable messages are sent synchronously.	false
connection-factory.block-on-durable-send	Boolean	Whether or not durable messages are sent synchronously.	true
connection-factory.call-timeout	Long	The timeout (in ms) for remote calls.	30000
connection-factory.client-failure-check-period	Long	The period (in ms) after which the client will consider the connection failed after not receiving packets from the server.	5000
connection-factory.client-id	String	The pre-configured client ID for the connection factory.	null
connection-factory.connection-load-balancing-policy-class-name	String	The name of the load balancing class. The default is: **`org.hornetq.api.core.client.loadbalance.roundRobinConnectionLoadBalancingPolicy`**	See description
connection-factory.connection-ttl	Long	the time to live (in ms) for connections	1 * 60000

Element Name	Type	Description	Default
connection-factory.consumer-max-rate	Integer	The fastest rate a consumer may consume messages per second.	-1
connection-factory.consumer-window-size	Integer	The window size (in bytes) for consumer flow control.	1024 * 1024
connection-factory.dups-ok-batch-size	Integer	The batch size (in bytes) between acknowledgments when using **`DUPS_OK_ACKNOWLEDGE`** mode.	1024 * 1024
connection-factory.failover-on-initial-connection	Boolean	Whether or not to failover to backup on event that initial connection to live server fails.	false
connection-factory.failover-on-server-shutdown	Boolean	Whether or not to failover on server shutdown.	false
connection-factory.min-large-message-size	Integer	The size (in bytes) before a message is treated as large.	100 * 1024
connection-factory.cache-large-message-client	Boolean	If true clients using this connection factory will hold the large message body on temporary files.	false
connection-factory.pre-acknowledge	Boolean	Whether messages are pre acknowledged by the server before sending.	false
connection-factory.producer-max-rate	Integer	The maximum rate of messages per second that can be sent.	-1
connection-factory.producer-window-size	Integer	The window size in bytes for producers sending messages.	1024 * 1024
connection-factory.confirmation-window-size	Integer	The window size (in bytes) for reattachment confirmations.	1024 * 1024
connection-factory.reconnect-attempts	Integer	Maximum number of retry attempts, -1 signifies infinite.	0
connection-factory.retry-interval	Long	The time (in ms) to retry a connection after failing.	2000
connection-factory.retry-interval-multiplier	Double	Multiplier to apply to successive retry intervals.	1.0
connection-factory.max-retry-interval	Integer	The maximum retry interval in the case a retry-interval-multiplier has been specified.	2000
connection-factory.scheduled-thread-pool-max-size	Integer	The size of the *scheduled thread* pool.	5
connection-factory.thread-pool-max-size	Integer	The size of the thread pool.	-1

Element Name	Type	Description	Default
connection-factory.transaction-batch-size	Integer	The batch size (in bytes) between acknowledgments when using a transactional session.	1024 * 1024
connection-factory.use-global-pools	Boolean	Whether or not to use a global thread pool for threads.	true
queue	Queue	A queue to create and add to JNDI.	
queue.name (attribute)	String	Unique name of the queue.	
queue.entry	String	Context where the queue will be bound in JNDI (there can be many).	
queue.durable	Boolean	Is the queue durable?	true
queue.filter	String	Optional filter expression for the queue.	
topic	Topic	A topic to create and add to JNDI.	
topic.name (attribute)	String	Unique name of the topic.	
topic.entry	String	Context where the topic will be bound in JNDI (there can be many).	

Appendix A. Revision History

Revision 5.1.1-100 **Mon Jul 18 2011** **Jared Morgan** *jmorgan@redhat.com*

Incorporated changes for JBoss Enterprise Application Platform 5.1.1 GA. For information about documentation changes to this guide, refer to *Release Notes 5.1.1*.